A Design Space Exploration Methodology for Enabling Tensor Train Decomposition in Edge Devices

Milad Kokhazadeh[1], Georgios Keramidas[1,2],

Vasilios Kelefouras[3], and Iakovos Stamoulis[2]

[1] School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
kokhazad@csd.auth.gr, gkeramidas@csd.auth.gr
[2] Think Silicon, S.A. An Applied Materials Company, Patras, Greece
g.keramidas@think-silicon.com, i.stamoulis@think-silicon.com
[3] School of Engineering, Computing and Mathematics, University of Plymouth, Plymouth, United Kingdom
vasilios.kelefouras@plymouth.ac.uk

Abstract. Deep Neural Networks (DNN) have made significant advances in various fields, including speech recognition and image processing. Typically, modern DNNs are both compute and memory intensive and as a consequence their deployment on edge devices is a challenging problem. A well-known technique to address this issue is Low-Rank Factorization (LRF), where a weight tensor is approximated with one or more lower-rank tensors, reducing the number of executed instructions and memory footprint. However, finding an efficient solution is a complex and time-consuming process as LRF includes a huge design space and different solutions provide different trade-offs in terms of FLOPs, memory size, and prediction accuracy. In this work a methodology is presented that formulates the LRF problem as a (FLOPs vs. memory vs. prediction accuracy) Design Space Exploration (DSE) problem. Then, the DSE space is drastically pruned by removing inefficient solutions. Our experimental results prove that it is possible to output a limited set of solutions with better accuracy, memory, and FLOPs compared to the original (non-factorized) model. Our methodology has been developed as a standalone, parameterized module integrated into T3F library of TensorFlow 2.X.

Keywords: Deep Neural Networks, Network Compression, Low-Rank Factorization, Tensor Train, Design Space Exploration.

1 Introduction

In recent years, the world has witnessed the era of Artificial Intelligence (AI) revolution, especially in the fields of Machine Learning (ML) and Deep Learning (DL), attracting the attention of many researchers in various applications fields [1]. Such an example is the Internet-of-Things (IoT) ML-based applications, where DNNs are employed on embedded devices with limited compute and memory capabilities [2].

State-of-the-art DNN models consist of a vast number of parameters (hundreds of billions) and require trillions of computational operations not only during the training, but also at the inference phase [3]. Therefore, executing these models on Resource-Constrained Devices (RCD), e.g., edge and IoT devices, is a challenging task. The problem becomes even more challenging when the target applications are characterized by specific real-time constraints [4]. As a result, many techniques have been recently proposed to compress and accelerate DNN models [5, 6]. DNN compression methods reduce the model's memory size and computational requirements without significantly impacting its accuracy. In general, DNN compression techniques are classified into five main categories [6]: pruning [7], quantization [8], compact convolutional filters [9], knowledge distillation [10], and low-rank factorization [11].

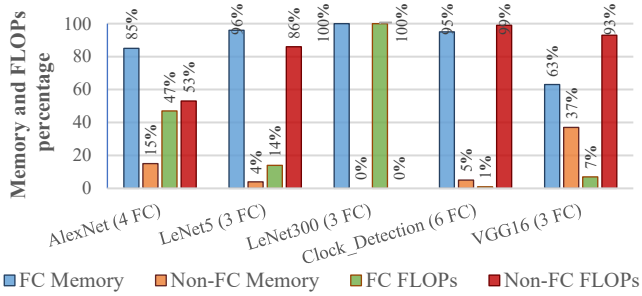


Fig. 1. Memory and FLOPs percentages of FC and Non-FC layers for the five different models considered in this work. FC layers take up a large portion of a DNN model's memory (from 63% up to 100 %)

Pruning techniques reduce the complexity of DNN models by removing unnecessary elements [7] (e.g., neurons or filters [12-14]). Quantization is a well-studied technique targeting to transform the 32-bit Floating-Point (FP32) weights and/or activations into less-accurate data types e.g., INT8 or FP16 [8]. In the compact convolutional filter techniques, special structural convolutional filters are designed to reduce the parameter space and save storage/computations [9]. Finally, knowledge distillation is the process of transferring the knowledge from a large model to a smaller one by following a student-teacher learning model [15]. The two latter techniques can be applied only to the convolutional layers [6]. On the contrary, Low-Rank Factorization (LRF) can be used to reduce both the number of Floating Point Operations (FLOPs) as well as the memory size in both convolutional and Fully Connected (FC) layers by transforming the original tensors into smaller ones [11]. However, employing LRF in a neural network model is not trivial: it includes a huge design space and different solutions provide different trade-offs among FLOPs, memory size, and prediction accuracy; therefore, finding an efficient solution is not a trivial task.

In this paper, we provide a methodology to guide the LRF process focusing on the FC layers of typical DNN models, as the FC memory arrays typically account for the largest percentage of overall memory size of DNN models. **Fig. 1** indicates that the memory size of FC layers' ranges from 63% up to 100% of the total DNN memory size.

The steps of the proposed methodology are as follows. First, all the FC layers' parameters are extracted from a given DNN model. Second, all possible LRF solutions are generated using the T3F library [16]. Then, the vast design space is pruned in two phases and a (limited) set of solutions is selected for re-evaluation/calibration according to specific target metrics (FLOPs, memory size, and accuracy) that are provided as inputs to our methodology.

The main contributions of this work are:

- A method that formulates the LRF problem as a (FLOPs, memory size, accuracy) DSE problem
- A step-by-step methodology that effectively prunes the design space
- A fully parameterized and standalone DSE tool integrated into T3F library (part of TensorFlow 2.X [17])
- An evaluation of the proposed DSE approach on five popular DNN models

The rest of this paper is organized as follows: In Section 2, we put this work in the context of related work and present the relevant background information. The proposed methodology is presented in a step-by-step basis in Section 3, while the experimental results are discussed in Section 4. Finally, Section 5 is dedicated to conclusions and future work.

2 Background and Related Works

2.1 Low-Rank Factorization

LRF refers to the process of approximating and decomposing a matrix or a tensor by using smaller matrices or tensors [18]. Suppose $M \in \mathbb{R}^{m \times n}$ is a matrix with m rows and n columns. Given a rank r, M can be approximated by $M' \in \mathbb{R}^{m \times n}$; M' has a lower rank k and it can be represented by the product of two (or more) thinner matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{k \times n}$ with m rows/k columns and k rows/n columns, respectively (as shown in **Fig. 2**). The element *(i,j)* from M is retrieved by multiplying the *i-th* row of U by the *j-th* column of V. The original matrix M needs to store $m \times n$ elements, while the approximated matrix M' needs to store $(m \times k) + (k \times n)$ elements [18].

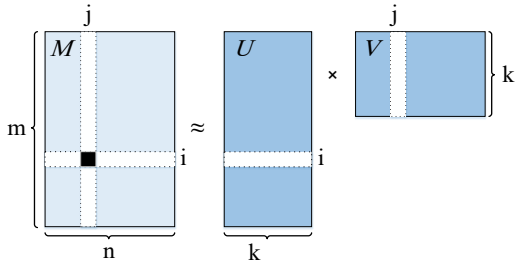


Fig. 2. Low-Rank Factorization (LRF)

To decompose the input matrices, different methods exist, such as Singular Value Decomposition (SVD) [19-21], QR decomposition [22, 23], interpolative decomposition

[24], and none-negative factorization [25]. Given that tensors are multidimensional generalizations of matrices, they need different methods to be decomposed e.g., Tucker Decomposition [26, 27] or Canonical Polyadic Decomposition (CPD) [28, 29]. Another way to decompose tensors is to transform the input tensor into a two-dimensional (2D) matrix and then perform the decomposition process using one of the abovementioned matrix decomposition techniques [30, 31].

2.2 Tensor-Train (TT) Format and T3F Library

A popular method to decompose the multidimensional tensors is the TT format, proposed in [32]. This is a stable method and does not suffer from curse of dimensionality [32]; furthermore, the number of parameters needed is similar to that in CPD [32]. A Tensor $A(j_1, j_2, ..., j_d)$ with d dimensions can be represented in TT format if for each element with index $j_k = 1, 2, ..., n_k$ and each dimension $k = 1, 2, ..., d$ there is a collection of matrices $G_k[j_k]$ such that all the elements of $\boldsymbol{A}$ can be computed by the following product [33]:

$$A(j_1.j_2.\ldots.j_d) = G_1[j_1]G_2[j_2] \ldots G_d[j_d] \quad (1)$$

All the matrices $G_k[j_k]$ related to the same dimension k are restricted to be of the same size $r_{k-1} \times r_k$. The values r_0 and r_d are equal to 1 in order to keep the matrix product (eq. 1) of size 1x1.

As noted, the proposed DSE methodology is built on top of the T3F library as a fully parameterized, stand-alone module. T3F is a library [16] for TT-Decomposition and currently is only available for FC layers (however our methodology is general enough and can be applied also in convolution layers; however, extending the proposed methodology in convolution layers is left for future work). In the current version, our target is the FC layers, because as depicted in **Fig. 1**, the FC layers occupy the largest memory size in typical DNN architectures. The main primitive of T3F library is *TT-Matrix*, a generalization of the Kronecker product [34]. By using the *TT-format,* T3F library compresses the 2D array of a FC layer by using a small set of parameters.

The inputs to the modified T3F module are: i) the weight matrix of the original FC layer (2D array), ii) the max_tt_rank value, and iii) a set of tensor configuration parameters. The latter set of parameters is related to the shape of the output tensors; if these tensors are multiplied by each other, then the original 2D matrix can be approximated e.g., the following set of parameters [[7, 4, 7, 4], [5, 5, 5, 5]] approximates a matrix of size 784x625. In other words, by multiplying the first set of numbers ([7, 4, 7, 4]), the first dimension of weight matrix (784) is produced and by multiplying the second set of numbers ([5, 5, 5, 5]), the second dimension of weight matrix (625) is generated. The max_tt_rank parameter defines the density of the compression; small max_tt_rank values offer higher compression rate. After the decomposition is done, T3F library outputs a set of 4D tensors (called cores) with the following shapes/parameters sizes:

Core #1 dim: $(1, s_1, o_1, \max\_tt\_rank)$

Core #2 dim: $(\max\_tt\_rank, s_2, o_2, \max\_tt\_rank)$

Core #3 dim: $(\max\_tt\_rank, s_3, o_3, \max\_tt\_rank)$

Core #4 dim: $(\max\_tt\_rank, s_4, o_4, 1)$

Where s_1, s_2, s_3, s_4 and o_1, o_2, o_3, o_4 are related to the set of tensor configuration parameters ($[[s_1,s_2,s_3,s_4],[o_1,o_2,o_3,o_4]]$). As noted, if the above tensors are multiplied by each other, the original 2D matrix is approximated. The overall memory size needed is given by the following formula (2):

$$(1\times s_1\times o_1\times \max\_tt\_rank)+(\max\_tt\_rank\times s_2\times o_2\times \max\_tt\_rank)+$$
$$(\max\_tt\_rank\times s_3\times o_3\times \max\_tt\_rank)+(\max\_tt\_rank\times s_4\times o_4\times 1) \quad (2)$$

2.3 Motivation

The main challenge in employing LRF for a specific DNN model is to select a suitable rank parameter [31]. Given a predefined rank value, the process of extracting the decomposed matrices or tensors is a well-defined and straightforward process. For example, for an input matrix *M* and for a given rank value *r*, the (output) low rank matrix with the minimal or target approximation error can be generated by employing the SVD algorithm [35].

While many researchers devised techniques targeting to find the best rank value [31, 36-38], it has been proven that this is an NP-hard problem [31]. For example, assuming a max rank value of 10 in LeNet5 model [39] (LeNet5 consists of three FC layers with dimensions 400x120, 120x84, and 84x10, respectively), the entire design space contains about 252 million possible ways to configure the decomposed matrices. Given that a model calibration phase (typically for more than three epochs) must be employed for each extracted solution, this means that 252Mx3x1 seconds (approximately 8750 days assuming that each epoch takes about 1 second) are needed to cover the whole design space. To the best our knowledge there is no similar work that formulates the LRF problem as a DSE problem.

3 Methodology

To address the above problem, a DSE methodology and a fully parameterized tool are proposed in this work. The target is to ease and guide the LRF process in FC layers. The main steps of the proposed methodology are shown in **Fig. 3**. In the rest of this section, a detailed description of each step in **Fig. 3** is provided.

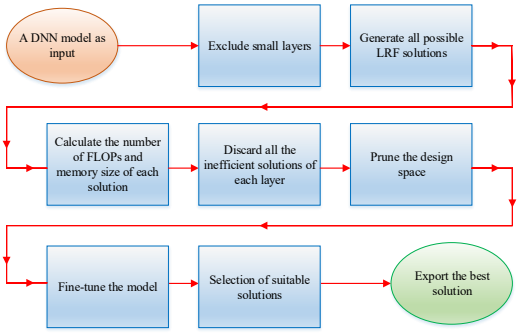


Fig. 3. The proposed DSE methodology

1. Exclude small layers: As noted, the first step of our approach is to extract and analyze all FC layers of a given DNN model. Among all the extracted FC layers, the layers with small memory sizes with respect to the overall memory footprint of the model (based on a threshold) are discarded. The aim of LFR is to reduce the memory size and computations required, thus applying LRF to layers with meager sizes does not provide significant memory/FLOP gains. As part of this work, a threshold with value equal to 10% (found experimentally) is used i.e., the layers of which the memory size is less than 10% of the overall DNN size are not considered in our methodology. Further quantifying this threshold is left for future work.
2. Generate all possible LRF solutions: In this step, all different LRF solutions are extracted using our methodology and generated by using the T3F library. For all remaining cases, the weight matrices are converted to smaller size tensors according to TT format. Note that in our approach, each solution (related to a weight matrix) is extracted as a set of configuration parameters. The first one is the length (based on the numbers participating in the composition e.g., for 100 = 2 * 5 * 5 * 2, the combination length is equal to 4). The second number is the maximum rank. The maximum rank defines how dense the compression will be. In this step, all possible solutions are extracted for each layer individually.

Let us give an example by using the well-known AlexNet model [40]. There are four FC layers in the AlexNet model; two of which are excluded by step 1. The remaining layers are of size 4096x4096 and 4096x1000, respectively. In both layers, 11 different rank values are included: {2, 3, …..., 12}. **Fig. 4** depicts all different solutions for the first layer. As we can see, there are 7759741 different solutions for this layer.

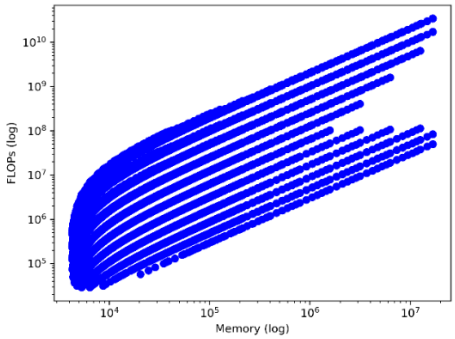


Fig. 4. All possible solutions based on T3F library for a layer of size 4096x4096 in the AlexNet model. Vertical axis shows the number of FLOPs (log scale) and horizontal axis shows the number of parameters (log scale)

3. Calculate the number of FLOPs and memory size for each solution: The mathematical expressions to calculate the required memory and FLOPs for each solution are given by equations (3) and (4), respectively. More specifically, assuming a FC layer with shape *[X, Y]*, the memory size is given by the following formula:

$$Memory\_required = ((\sum_{i=1}^{L}\prod_{j=1}^{4} c_{i,j}) + Y) \times Type\_size \quad (3)$$

Where L is amount of cores/length of combination, $c_{i,j}$ is *j-th* element in the *i-th* core, and Y is the length of bias vector. The LRF takes as input a 2D array and generates a number of 4D tensors (called also cores). To calculate the memory size required for a layer, we need first to calculate and sum up the number of parameters for each core. Then, the bias must be added to the calculated value. Finally, to find the required memory, the number of parameters is multiplied by the number of bytes for the used data type (e.g., 4 bytes for FP).

The number of FLOPs is given by the following formula:

$$FLOPs = (\sum_{i=L}^{1}(cl_i \times cr_i \times inr_i + (\prod_{j=1}^{4} c_{i,j} + inl_i \times inr_i))) + Y \quad (4)$$

Where $inr_i = \prod_{j=1}^{L-1} x_j$, $inl_i = c_{i,2} \times c_{i,4}$, $cr_i = c_{i,2} \times c_{i,4}$, $cl_i = c_{i,1} \times c_{i,3}$ and x_j is *j-th* element in the input combination.

4. Discard inefficient solutions of each layer: As mentioned above, the design space is vast, therefore fine-tuning or calibrating the model for all possible solutions is not feasible. To address this, the whole design space (illustrated as a FLOPs vs. memory size pareto curve) is divide into four rectangles (see **Fig. 5**). The top-right part (red part in **Fig. 5**) is excluded for the remaining steps, since it contains solutions that require more memory and more FLOPs compared to the non-factorized (initial) solution. Note that the blue dot in the center of the graph corresponds to the memory/FLOPs of the initial layer. The top-left and bottom-right parts (yellow parts) are also excluded in the current version of our methodology. Although the latter two parts can contain acceptable solutions, we have excluded them as the solutions in these parts require more FLOPs (top left) or more memory (bottom right) compared to the initial model. The bottom left part (green part) includes solutions that exhibit less memory and less FLOPs with respect to the initial layer. As part of this work, we consider only the solutions in the green box i.e., these solutions will be forwarded to the remaining steps of our methodology. The solutions included in the two yellow boxes will be re-considered in a future version of this work.

5. Prune the design space: Till now, we considered each layer separately. As a next step, we take into consideration all the FC layers of the input DNN model in a unified way. In particular, in the current step, the 2D design space (green part in **Fig. 5**) of each layer is further broken down into smaller rectangles (tiles) of predefined size. In this paper, an 8x8 grid is considered i.e., each green rectangle in **Fig. 5** is broken down into 64 tiles. Examining alternative configurations (e.g., 4x4 or 16x16 grids) is left for future work. The bottom-line idea is that the solutions residing into the same tile will exhibit a similar behavior. To safeguard the latter approach, multiple points (solutions) from each tile are extracted. More specifically, the four following solutions are considered from each tile:

- **Point 1:** min_FLOPs and min_memory
- **Point 2:** min_FLOPs and max_memory
- **Point 3:** max_FLOPs and min_memory

- **Point 4:** max_FLOPs and max_memory

The four solutions above include the best and worst solutions in terms of FLOPs and/or memory per tile; the solutions with highest FLOPs and memory (max_FLOPs, max_memory) are selected as it is more likely to provide higher prediction accuracy (after the re-calibration/training phase). Given that there are 64 tiles, a maximum number equal to 4x64 solutions are further processed for each layer and the rest are being discarded.

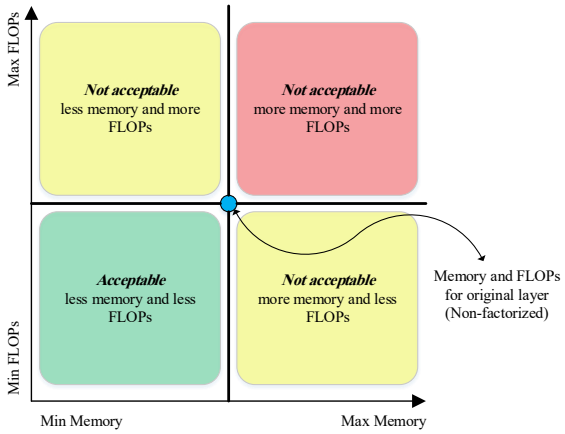
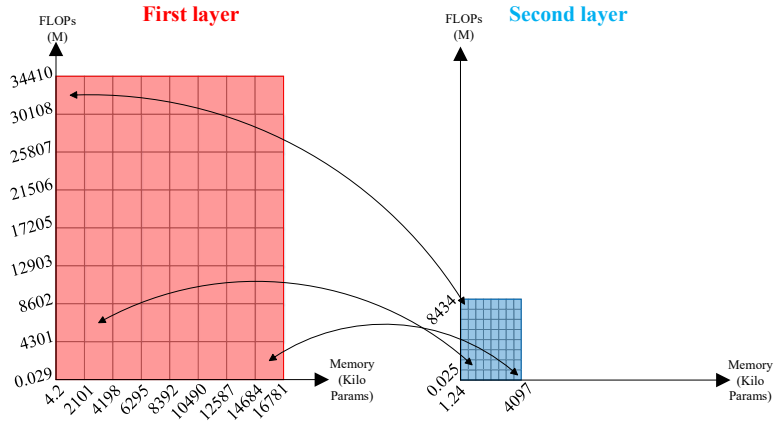


Fig. 5. The design space is illustrated as (FLOPs vs. memory) pareto curves and partitioned into rectangles. The red part and yellow parts are pruned (excluded) because they contain solutions with more memory and/or FLOPs compared to the original (non-factorized) layer

After the design space is pruned at a layer level, the corresponding tiles of all layers are merged and the four points mentioned above are extracted. However, when multiple FC layers co-exist in the model (which is typically the case), the approach illustrated in **Fig. 6** is followed. The main problem that we need to tackle at this point is that different layers might include tiles of different scales and consequently very diverse memory and FLOPs requirements. To address this in our methodology, each layer is organized as a separate grid in order to take into account the different scales (i.e., the size of each layer) as depicted in **Fig. 6**. Note that in the case of multiple FC layers, the corresponding grid cells must be selected for all layers. In the special case in which no solution exists in the corresponding grid tile (in one or more layers), the following two approaches are considered in our methodology: i) the nearest grid cell (to the empty cell) must be found and solutions from that cell are selected and ii) the grid cells that have at least one layer with no solution are skipped (excluded) and we only consider the grid cells where all layers contain at least one solution. This is illustrated in **Fig. 6**.

Let us give an example based on AlexNet model. Recall that in our methodology we consider two layers of AlexNet model with shapes 4096x4096 and 4096x1000 (shown in **Fig. 6)**. It is clear that for each grid cell in the first layer (**Fig. 6.a**), there is an unique corresponding grid cell in the second layer (**Fig. 6.b**). By relying in the second grid cell selection policy (mentioned above) for the empty cells, many solutions will be further

excluded and will not be considered in the following steps of the approach, further pruning the design space.



(a) (b)

Fig. 6. An 8x8 grid corresponding to the AlexNet model. Two FC layers have been selected from the model for factorization. The left part of the figure (a) depicts the first layer with shape 4096x4096 (16781K parameters) and the right part (b) shows the second layer with shape 4096x1000 (4097K parameters). For each grid cell in the first layer (a), there is a unique grid cell in the second layer (c)

6. Fine-tune the remaining solutions: After selecting different points from each tile, the next step is to calibrate (i.e., re-train) the model for the extracted solutions and for a limited number of epochs (e.g., three to five epochs). The output of the latter step is to accommodate each extracted solution with the following points: FLOPs, memory, and accuracy loss.

Table 1. Number of solutions after each step. After step 7, solutions with accuracy degradation more than 1.5 % are excluded

Method	**Parameters (M)**		**Number of Solutions**				**Dataset**
	Model	***FC (%)***	***Original***	***After step 4***	***After step 5***	***After step 7***	
LeNet5	0.062	0.059 (**95.8%**)	1528 M	157 M	156	150	MNIST
LeNet300	0.267	0.267 (**100%**)	1495 M	239 M	156	137	MNIST
VGG16	39.98	25.21 (**63%**)	7285832 G	1012815 G	144	43	CIFAR10
Alexet	25.73	21.94 (**85.3%**)	4327 G	2454 G	140	135	CIFAR10
Clock_Detection	0.96	0.912 (**94.9%**)	16 G	1708 M	256	215	Self-generated

7. Selection of suitable solutions: The next and final step is to go through the output solutions and produce the final output based on specific high-level criteria. This means that after step 6 is completed (calculation of loss and accuracy for the remaining solutions), we can enforce specific constrains (set by the user or the application) in terms

of memory footprint reduction, FLOPs reduction, and/or accuracy loss. Note that our approach is fully parameterized and any kind of high-level criteria can be employed (e.g., to exclude the solutions that have >1.5% accuracy drop). **Table 1** shows in details the initial number of LRF solutions and the solutions that are pruned in each step of the proposed methodology for the five models that we consider in this work.

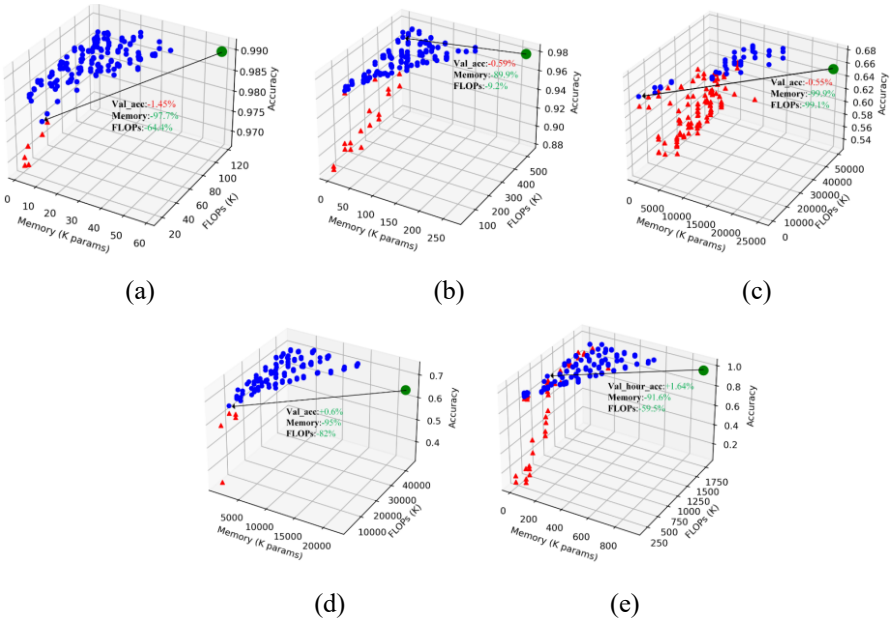


Fig. 7. 3D pareto curves (memory footprint, FLOPs, and accuracy) for the five studied models: (a) LeNet5, (b) LeNet300, (c) VGG16, (d) AlexNet, and (e) Clock_Detection. Green circles correspond to the non-factorized model; blue circles indicate the solutions with accuracy drop less than 1.5%; red triangles show the solutions with accuracy drop more than 1.5%

4 Experimental Results

As noted our evaluation is based on multiple datasets and DNN models: LeNet300 and LeNet5 on MNIST dataset; AlexNet and VGG16 on CIFAR10 dataset; and Clock Detection model on self-generated data. In all cases, we compare our experimental results to the baseline (not factorized model). All experiments are initialized from reference models that we have developed from scratch and train for 100 epochs. For each compressed model, we report its validation accuracy, storage (number of parameters), and FLOPs. We calculate FLOPs based on the assumption that each multiplication or addition are considered as one FLOPs. For example, in a forward pass through a FC layer with weight matrix of m × n and bias of n × 1, the considered FLOPs are 2 × m × n + n. We use an 8x8 grid for all cases and we exclude the solutions with >1.5% accuracy drop compared to the initial model. The final results are shown in **Fig. 7** as 3D pareto curve (memory, FLOPs, and accuracy) for each evaluated model.

In each graph in **Fig. 7**, the green circles correspond to the non-factorized model, the blue circles are referred to the solutions with accuracy drop less than 1.5% with respect to the initial model, and the red triangles are referred to the solutions with accuracy drop more than 1.5%. In addition, the extracted solution with the lowest memory footprint is annotated with the black arrow in all cases. Our experimental results on MNIST dataset show that we managed to achieve a 97.7% memory reduction in the LeNet5 model (**Fig. 7.a**) and 89.9% memory reduction in the LeNet300 model (**Fig. 7.b**) with only 1.45% and 0.59% accuracy drop, respectively. Similar results can be seen in the other models as well.

Table 2. Example solutions extracted from our methodology (numbers with green colors represent a reduction in memory footprint or number of flops or an increase in model accuracy; number with red colors correspond to accuracy drop). All numbers are normalized to the corresponding parameters of the initial model

Models (initial parameters)	**Memory reduction**	**FLOPs reduction**	**Accuracy**	**Example Solutions**
LeNet5 (Acc.=0.99, Mem.=58284, FLOPs=116364)	40.6%	11.3%	0.2%	*highest accuracy*
	97.7%	64.4%	-1.45%	*lowest memory*
	90.4%	86.2%	-0.4%	*lowest FLOPs*
LeNet300 (Acc.=0.979, Mem.=265600, FLOPs=530800)	84%	10.7%	0.1%	*highest accuracy*
	89.9%	9.2%	-0.59%	*lowest memory*
	88.5%	84.6%	-1.3%	*lowest FLOPs*
VGG 16 (Acc.=0.653, Mem.=33562624, FLOPs=50339840)	62.5%	25%	3.33%	*highest accuracy*
	99.9%	99.1%	-0.55%	*lowest memory*
	99.9%	99.1%	-0.55%	*lowest FLOPs*
AlexNet (Acc.=0.6408, Mem.=20878312, FLOPs=41751528)	62%	13%	13%	*highest accuracy*
	95%	82%	0.6%	*lowest memory*
	87%	83.9%	8.9%	*lowest FLOPs*
Clock_Detection (Acc.=0.97, Mem.=907400, FLOPs=1814600)	74.5%	61.5%	2.88%	*highest accuracy*
	91.6%	59.5%	1.64%	*lowest memory*
	91.5%	87%	-1.14%	*lowest FLOPs*

For clarity reasons, **Table 2** present three specific example cases for all the models considered in this work. The three cases correspond to the solutions with: highest reported accuracy, lowest memory, and lowest FLOPs. As **Table 2** indicates, our methodology is able to extract solutions exhibiting a reduction in memory footprint from 40.6% (in LeNet5) up to 99.9% (in VGG16) and a reduction in number of FLOPs from 9.2% (in LeNet300) up to 99.1% (in VGG16). Finally, it is important to note, that in many cases, our approach manages not only to reduce the memory footprint and the number of FLOPs, but also to increase the prediction accuracy of the specific models. More specifically, an increase in the accuracy is reported in eight out the 15 cases (up to 13% increase in AlexNet) as shown in **Table 2**.

5 Conclusion

In this paper, we presented a practical methodology that formulates the compression problem in DNN models using LRF as a DSE problem. The proposed methodology is

able to extract a suitable set of solutions in a reasonable time. We evaluated our methodology on five different DNN models. Our experimental findings revealed that the proposed approach can offer a wide range of solutions that are able to compress the DNN models up to 97.7% with minimal impact in accuracy. Part of our future work includes the investigation of the additional techniques to further prune the design space. In addition, we plan to employ our methodology to different types of NN layers such as convolution layers. Finally, we also plan to extent and customize our methodology to NN belonging to different application areas, such as object detection, image segmentation, text and video processing.

Acknowledgements

This research has been supported by the H2020 Framework Program of the European Union through the Affordable5G Project (Grant Agreement 957317) and by a sponsored research agreement between Applied Materials, Inc. and Aristotle University of Thessaloniki, Greece (Grant Agreement 72714).

References

1. Fatima Hussain, Rasheed Hussain, Syed Ali Hassan, and Ekram Hossain. Machine learning in IoT security: Current solutions and future challenges. Communications Surveys & Tutorials, 2020.
2. Surbhi Saraswat, Hari Prabhat Gupta, and Tanima Dutta. A writing activities monitoring system for preschoolers using a layered computing infrastructure, Sensors Journal, 2019.
3. Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. arXiv preprint arXiv:2104.08378, 2021.
4. Ayten Ozge Akmandor, Y. I. N. Hongxu, and Niraj K. Jha. Smart, secure, yet energy-efficient, Internet-of-Things sensors. Transactions on Multi-Scale Computing Systems, 2018.
5. Xin Long, Zongcheng Ben, and Yan Liu. A survey of related research on compression and acceleration of deep neural networks. Journal of Physics: Conference Series, 2019.
6. Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. arXiv preprint arXiv:1710.09282, 2017.
7. Morteza Mousa Pasandi, Mohsen Hajabdollahi, Nader Karimi, and Shadrokh Samavi. Modeling of pruning techniques for deep neural networks simplification. arXiv preprint arXiv:2001.04062, 2020.
8. Zhuoran Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. Drq: dynamic region-based quantization for deep neural network acceleration. Intl. Symposium on Computer Architecture, 2020.
9. Fuxiang Huang, Lei Zhang, Yang Yang, and Xichuan Zhou. Probability weighted compact feature for domain adaptive retrieval. Intl. Conference on Computer Vision and Pattern Recognition, 2020.
10. Cody Blakeney, Xiaomin Li, Yan Yan, and Ziliang Zong. Parallel blockwise knowledge distillation for deep neural network compression. IEEE Transactions on Parallel and Distributed Systems, 2020.

11. Anh-Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavský, Valeriy Glukhov, Ivan Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. European Conference on Computer Vision, 2020.
12. Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. arXiv preprint arXiv:1808.06866, 2018.
13. Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. Intl. Conference on Computer Vision, 2017.
14. Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. arXiv preprint arXiv:1506.02626, 2015.
15. Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. Knowledge distillation: A survey. Journal of Computer Vision, 2021.
16. Alexander Novikov, Pavel Izmailov, Valentin Khrulkov, Michael Figurnov, and Ivan V. Oseledets. Tensor Train Decomposition on TensorFlow (T3F). Journal of Machine Learning, 2020.
17. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin et al. Tensorflow: A system for large-scale machine learning. USENIX Symposium on Operating Systems Design and Implementation, 2016.
18. Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. Intl. Conference on Acoustics, Speech and Signal Processing, 2013.
19. Jiong Zhang, Qi Lei, and Inderjit Dhillon. Stabilizing gradients for deep neural networks via efficient svd parameterization. Intl. Conference on Machine Learning, 2018.
20. Mohammad Mahdi Bejani, and Mehdi Ghatee. Theory of adaptive SVD regularization for deep neural networks. Journal of Neural Networks, 2020.
21. Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. Sparse low rank factorization for deep neural network compression. Journal of Neurocomputing, 2020.
22. Arsenia Chorti, and David Picard. Rate analysis and deep neural network detectors for SEFDM FTN systems. arXiv preprint arXiv:2103.02306, 2021.
23. Iordan Ganev, and Robin Walters. The QR decomposition for radial neural networks. arXiv preprint arXiv:2107.02550, 2021.
24. Jerry Chee, Megan Renz, Anil Damle, and Chris De Sa. Pruning neural networks with interpolative decompositions. arXiv preprint arXiv:2108.00065, 2021.
25. Teck Kai Chan, Cheng Siong Chin, and Ye Li. Non-negative matrix factorization-convolutional neural network (NMF-CNN) for sound event detection. arXiv preprint arXiv:2001.07874, 2020.
26. Dawei Li, Xiaolong Wang, and Deguang Kong. Deeprebirth: Accelerating deep neural network execution on mobile devices. Conference on Artificial Intelligence, 2018.
27. Zongwen Bai, Ying Li, Marcin Woźniak, Meili Zhou, and Di Li. Decomvqanet: Decomposing visual question answering deep network via tensor decomposition and regression. Journal of Pattern Recognition, 2021.
28. Frusque Gaetan, Michau Gabriel, and Fink Olga. Canonical Polyadic Decomposition and Deep Learning for Machine Fault Detection. arXiv preprint arXiv:2107.09519, 2021.
29. Ruixin Ma, Junying Lou, Peng Li, and Jing Gao. Reconstruction of Generative Adversarial Networks in Cross Modal Image Generation with Canonical Polyadic Decomposition. Wireless Communications and Mobile Computing Conference, 2021.
30. Tamara G. Kolda, and Brett W. Bader. Tensor decompositions and applications. Journal of SIAM review, 2009.

31. Yerlan Idelbayev, and Miguel A. Carreira-Perpinán. Low-rank compression of neural nets: Learning the rank of each layer. Conference on Computer Vision and Pattern Recognition, 2020.
32. Ivan V. Oseledets. Tensor-train decomposition. SIAM Journal on Scientific Computing, 2011.
33. Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. arXiv preprint arXiv:1509.06569, 2015.
34. D. Pollock and G. Stephen. Multidimensional Arrays, Indices and Kronecker Products. Journal of Econometrics, 2021.
35. Gene H. Golub and Charles F. Van Loan. Matrix computations. Jonhs Hopkins University Press, 2013.
36. Cole Hawkins, Xing Liu, and Zheng Zhang. Towards Compact Neural Networks via End-to-End Training: A Bayesian Tensor Approach with Automatic Rank Determination. arXiv preprint arXiv:2010.08689, 2020.
37. Zhiyu Cheng, Baopu Li, Yanwen Fan, and Yingze Bao. A novel rank selection scheme in tensor ring decomposition based on reinforcement learning for deep neural networks. Intl. Conference on Acoustics, Speech and Signal Processing, 2020.
38. Taehyeon Kim, Jieun Lee, and Yoonsik Choe. Bayesian optimization-based global optimal rank selection for compression of convolutional neural networks. IEEE Access, 2020.
39. Yann LeCun. LeNet-5, convolutional neural networks. url: http://yann.lecun.com/exdb/lenet/, 2015.
40. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. Communications of the ACM, 2017.