Automated Multi-Stage Compression of Neural Networks

Julia Gusak[1*],   Maksym Kholiavchenko[2*],   Evgeny Ponomarev[1],   Larisa Markeeva[1],
Philip Blagoveschensky[1,3],   Andrzej Cichocki,[1]   Ivan Oseledets[1]

[1]Skolkovo Institute of Science and Technology, Russia
[2]Innopolis University, Russia   [3]Huawei Noah's Ark Lab
m.kholyavchenko@innopolis.ru {*y.gusak, evgenii.ponomarev, l.markeeva,*
philip.blagoveschensky, a.cichocki, i.oseledets}@skoltech.ru

Abstract

Low-rank tensor approximations are very promising for compression of deep neural networks. We propose a new simple and efficient iterative approach, which alternates low-rank factorization with smart rank selection and fine-tuning. We demonstrate the efficiency of our method comparing to non-iterative ones. Our approach improves the compression rate while maintaining the accuracy for a variety of tasks.

1. Introduction

The development of deeper and more complex networks in order to achieve higher performance has become commonplace. However such networks contain tens of millions of parameters and often cannot be efficiently deployed on embedded systems and mobile devices due to their computational power and memory limitations.

Low-rank matrix and tensor approximations provide excellent compression of neural network layers [12, 13, 14, 25]. In these methods, factorization of weight tensors yields an approximate compressed network. For example, when we approximate a 4-dimensional convolutional kernel by a tensor, whose factorized form has three components, we can replace a corresponding layer with three consecutive convolutional layers (Figure 3). However, approaches based on a low-rank tensor factorization are built on the same scheme: compression followed by fine-tuning to compensate for a significant loss of the quality of the model. The main benefit of this approach is that the compressed version provides an initial approximation, which leads to a better quality after fine-tuning than if the same architecture is learned directly from random initialization.

In our paper, we propose a way to substantially improve the abovementioned scheme by applying low-rank decompo-

sition and fine-tuning iteratively several times (Algorithm 1). It turns out that such a simple idea can significantly improve the quality of neural networks compression. For example, for Faster R-CNN with ResNet-50 backbone, we achieve a better-compressed model for the same quality than can be obtained with a non-iterative algorithm (Section 6).

We introduce *Multi-Stage COmpression method (MUSCO)* for automated network compression (Sections 2, 3). The algorithm consists of two repetitive steps: compression and fine-tuning. During the compression step model weights of selected layers are approximated using tensor decomposition with automatically selected rank values (Section 5). At this step, the redundancy present in the weight parameters is partially reduced. The next step allows for recovering the quality of the model by performing fine-tuning. By repeating these two steps several times, we can gradually compress the model by substantially reducing the number of parameters in the selected layers.

In comparison with other approaches, MUSCO does not loose quality significantly during more aggressive parameters reduction. In practice, MUSCO allows achieving higher compression ratio than state-of-the-art non-iterative methods with the same quality of the model. Our main contributions are:

- We propose an iterative low-rank approximation algorithm to efficiently compress neural networks that outperform non-iterative methods for the desired accuracy.

- We introduce a method for automatic tensor rank selection for tensor approximations performed at each compression step.

- We validate and demonstrate the high efficiency of our approach in a series of extensive computational experiments for object detection and classification problems. Source code is publicly available at https://github.com/juliagusak/musco.

---

*Contributed equally.

2. Problem statement

In this section, we introduce a formal description of a model compression in terms of transitions from one class of models to another.

Each neural network can be described as a pair (f, θ), where θ denotes model parameters and f defines network architecure (i.e. graph structure). Given θ, a continuous function f assigns to each input X a result of its propagation through the whole network, $f(X, \theta)$.

Let M be our pre-trained neural network model. We denote the class of all neural networks with the same architecture as M by $\mathcal{M} = \{(f, \theta) | \theta \in \Theta\}$. Here θ is an array of weight tensors that parametrize individual layers of the network architecture f, and Θ defines a set of all possible parametrizations.

We perform a network compression via low-rank tensor approximation of its weight tensors θ. The concept of *rank* can be defined for any tensor. We use $\text{rank}(\theta)$ to determine an array of ranks corresponding to tensors in θ. The expression $\text{rank}(\theta) \leq R$, where R is an array of constant values, is used to describe elementwise constraints applied to tensors from θ.

To apply a *rank-R factorization* to the weights $\theta \in \Theta$ is to find an array of $\hat{\theta}$ from Θ^R,

$$\Theta^R = \{\theta \in \Theta \mid \text{rank}(\theta) \leq R\}, \tag{1}$$

which approximates θ in a certain norm and can be represented in a factorized format $\hat{\theta}_{\text{fact}} \in \Theta^R_{\text{fact}}$ (i.e. $\hat{\theta}_{\text{fact}}$ is an array, where each element corresponds to one weight tensor from θ and represented by a tuple of factors).

We introduce operators $\mathcal{F}_{\text{fact}}$ and $\mathcal{F}_{\text{full}}$ that perform these mappings from Θ^R to Θ^R_{fact} and vice versa, i.e.

$$\Theta^R_{\text{fact}} = \{\mathcal{F}_{\text{fact}}(\theta) \mid \theta \in \Theta^R\} \tag{2}$$

and $\mathcal{F}_{\text{full}}(\mathcal{F}_{\text{fact}}(\theta)) = \theta$ for $\theta \in \Theta^R$.

When we compress a network $(f, \theta) \in \mathcal{M}$ using a rank-R weight factorization (Figure 1), firstly, we obtain a model $(f, \hat{\theta})$ with the same arhitecture projecting θ to the parameter set $\Theta^R \subseteq \Theta$. Secondly, we get a compressed model $(f^R, \hat{\theta}_{\text{fact}}) \in \mathcal{M}_R$ with a new architecture f^R by replacing $\hat{\theta}$ with its factorized version $\hat{\theta}_{\text{fact}}$, where

$$\mathcal{M}_R = \{(f^R, \theta) \mid \theta \in \Theta^R_{\text{fact}}\} \tag{3}$$

and f^R is a modification of f, which contains decomposed linear layers instead of original ones. A *decomposed layer* is a sequence of linear layers, each of which is represented by one factor (component) from the factorization of the original layer weight tensor.

After fine-tuning a network $(f^R, \hat{\theta}_{\text{fact}})$, we obtain a model $(f^R, \theta_{\text{fact}}) \in \mathcal{M}_R$, which attains a local minimum of the loss function calculated on training samples.
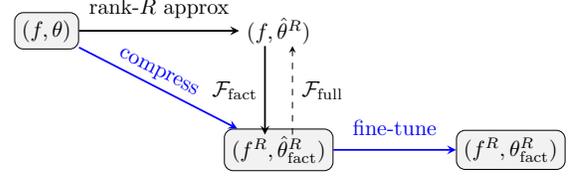

Figure 1. Iterative compression (first iteration): pre-trained weights θ are factorized and corresponding linear layers of the initial architecture f are decomposed into sequences of linear layers, which yield a new architecture f^R. (A number of layers in each sequence are equal to the number of components in the tensor factorization.)

When we further compress an already decomposed network $(f^R, \theta_{\text{fact}})$, we apply the mentioned rank-R' compression procedure to the model $(f, \theta) \in \mathcal{M}$, where $\theta \in \Theta^R$ (Figure 2). In Section 4 we show how this step can be optimized for different types of tensor factorizations.
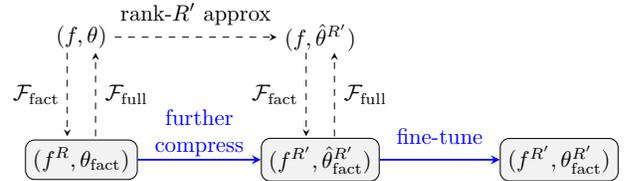

Figure 2. Iterative compression (next iteration): the rank of factorized weights θ_{fact} is further reduced. The architecture is changed, but the number of linear layers remains the same.

Thus, when we gradually compress the pre-trained model M over K iterations (each iteration contains compression and fine-tuning steps), we sequentially obtain models from classes $\mathcal{M}_{R_1}, \ldots, \mathcal{M}_{R_K}$, namely,

$$M \to \widehat{M}_1 \to M_1 \to \ldots \to \widehat{M}_K \to M_K, \tag{4}$$
$$\text{s.t.} \quad \widehat{M}_k, M_k \in \mathcal{M}_{R_k},$$

where $R_1 \geq \ldots \geq R_K$. Transitions $M_{k-1} \to \widehat{M}_k$ and $\widehat{M}_k \to M_k$ correspond, respectively, to the compression and fine-tuning steps, $k = 1 \ldots K$. To compare, for the non-iterative approach the similar path looks like $M \to \widehat{M}_K \to M_K$, i.e., architecture f^{R_K} of resulting compressed model is determined at the first (and the only) compression step.

If we instead train the architecture f^{R_K} from scratch, it is often impossible to achieve the quality comparable to the initial model M. If the non-iterative approach is applied and we compress M directly to the model from $\mathcal{M}_{R_K}$ (as it is done in [14, 13]), after fine-tuning, we end up with a good baseline. We show that our iterative approach (Algorithm 1) beats the baseline in terms of compression ratio while preserving accuracy. (Section 6).

We can also describe our approach in a different way taking into account that each model from $\mathcal{M}_{R_k}$, $k = 1, \ldots, K$, can be mapped onto a model from $\mathcal{M}$ using the operator $\mathcal{F}_{\text{full}}$ (Figures 1,2). Namely, for the initial

architecture f, we iteratively reduce the set of valid parameters, $\Theta \supset \Theta^{R_1} \supset \cdots \supset \Theta^{R_K}$, and we search for the best parameters values under the imposed constraints.

Thus, every time we approximate weights (i.e., project weights to the smaller parameter subspace), we take a step away from the local minimum on the loss surface. Due to the continuity of the model, the more we reduce the weight rank, the bigger is the step and, hence, the more difficult it is to get back to a local optimum during the subsequent fine-tuning (because of the non-convexity of the optimization problem). In the iterative approach, in contrast to the non-iterative, the ranks decrease smoothly and gradually, and that allows to obtain a higher model compression rate with the same quality drop.

Moreover, our approach allows to perform compression by automatically searching for the best rank values $R_k, k = 1, ..., K$ (see Section 5 for the details).

3. Compression algorithm

In this section, we formulate the optimization problems which need to be solved during one iteration of the compression algorithm, and we provide a detailed procedure for iterative neural network model compression.

3.1. One iteration of the algorithm

To compress a layer with a weight tensor θ, for the first time, given rank R we solve a problem of minimizing Frobenius norm, given rank R:

$$\min_{\theta_1^R, ..., \theta_N^R} ||\theta - \hat{\theta}^R||, \quad \text{such that}$$
$$\mathcal{F}_{\text{fact}}(\hat{\theta}^R) = \left(\theta_1^R, ..., \theta_N^R\right), \tag{5}$$

where $\theta_1^R, \ldots, \theta_N^R$ denote components in a factorized form of the tensor and define weights of N layers into which initial layer is decomposed during the rank-R factorization[1].

To further compress an already decomposed layer, we update its fine-tuned weights $\{\theta_n^R\}_{n=1}^N$. Namely, given rank $R' < R$, we solve the following minimization problem:

$$\min_{\theta_1^{R'}, ..., \theta_N^{R'}} ||\mathcal{F}_{\text{full}}(\theta_{\text{fact}}) - \hat{\theta}^{R'}||, \quad \text{such that}$$
$$\theta_{\text{fact}} = (\theta_1^R, ..., \theta_N^R), \tag{6}$$
$$\mathcal{F}_{\text{fact}}(\hat{\theta}^{R'}) = (\theta_1^{R'}, ..., \theta_N^{R'}),$$

where $\theta_1^{R'}, \ldots, \theta_N^{R'}$ denote updated weights (factor matrices) of the decomposed layer.

During the fine-tuning step, we minimize the loss function ℓ given training data $\{(X_j, Y_j)\}_{j=1}^J$, where X_j is an

---

[1] Factors from tensor decomposition are equal to weights of the decomposed layer up to the reshape operation.

input sample, and Y_j is a corresponding target value. Thus, we solve the following optimization problem:

$$\mathcal{L}(\theta) \to \min_{\theta \in \Theta_{\text{fact}}^R}, \quad \text{s.t.} \quad \mathcal{L}(\theta) = \sum_{j=1}^J \ell\left(f^R(X_j, \theta), Y_j\right), \tag{7}$$

where f^R is our compressed architecture and Θ_{fact}^R is the set of all possible model parameters.

3.2. Iterative procedure

Our proposed algorithm is an alternation of compression and fine-tuning steps with automatically selected ranks for the weights approximation (see Algorithm 1 for the details).

At the compression step of each iteration for each of the selected layers, we solve the optimization problem (5) if a layer has not been compressed yet, and the optimization problem (6) otherwise. The fine-tuning step is the same for all iterations.

Algorithm 1 Iterative low-rank approximation algorithm for automated network compression

---
Input: Pre-trained original model, M
Output: Fine-tuned compressed model, M^*.
1: $M^* \leftarrow M$
2: **while** desired compression rate is not attained or automatically selected ranks have not stabilized **do**

3:      $R \leftarrow$ automatically selected ranks for low-rank tensor approximations of convolutional and fully-connected weight tensors.

4:      $\widehat{M} \leftarrow$ (further) compressed model obtained from M by replacing layer weights with their rank-R tensor approximations.

5:      $M^* \leftarrow$ fine-tuned model $\widehat{M}$.
6: **end while**

---

4. Layer compression in details

In this paper, we focus on Tucker and HOSVD (High Order Singular Value Decomposition) decompositions [24]. Our framework can be used for other decompositions as well, particularly for CP[2] and SVD.

Since the definition of a tensor rank is not unified, we use a multilinear rank for Tucker (see definition below) and a CPD tensor rank for CP.

A *Tucker decomposition* of an N-dimensional tensor is a factorization into a small size *core tensor* and N *factor matrices*. For a convolutional kernel $\theta \in \mathbb{R}^{d \times d \times C_{\text{in}} \times C_{\text{out}}}$,

---

[2] CP is a canonical polyadic decomposition [6, 10]

with C_{in} input channels, C_{out} output channels, and a $d \times d$ spacial filter, it can be written as

$$\theta \approx \theta_C \times_{\text{h}} \theta_{\text{h}} \times_{\text{w}} \theta_{\text{w}} \times_{\text{in}} \theta_{\text{in}} \times_{\text{out}} \theta_{\text{out}}, \qquad (8)$$

where θ_C is a 4-dimensional core tensor, $\theta_{\text{h}}, \theta_{\text{w}}, \theta_{\text{in}}, \theta_{\text{out}}$ are matrices to be multiplied along each dimension of the core tensor. Symbols $\times_{\text{h}}, \times_{\text{w}}$ and $\times_{\text{in}}, \times_{\text{out}}$ denote multilinear products along spacial and channel dimensions, respectively.

If decomposition (8) holds exatcly, the *multilinear rank* of the tensor θ is defined as a tuple $(R_{\text{h}}, R_{\text{w}}, R_{\text{in}}, R_{\text{out}})$, where the n-th element, $n = 1 \ldots 4$, is a rank of the *dimension-n unfolding* of the tensor[3].

In convolutional kernels spacial dimensions usually are quite small. Therefore, similar to [13], we factorize only two channel related dimensions, i.e. apply *Tucker-2 decomposition*, which is a specific form of the Tensor Train decomposition [20]:

$$\theta \approx \theta_C \times_{\text{in}} \theta_{\text{in}} \times_{\text{out}} \theta_{\text{out}}. \qquad (9)$$

The corresponding multilinear rank equals $(d, d, R_{\text{in}}, R_{\text{out}})$, everywhere later we refer to it as $(R_{\text{in}}, R_{\text{out}})$.

4.1. First-time layer compression using Tucker-2 kernel approximation

Let $\hat{\theta} \in \mathbb{R}^{d \times d \times C_{\text{out}} \times C_{\text{in}}}$ be a kernel approximation obtained via Tucker-2 decomposition with rank $R = (R_{\text{out}}, R_{\text{in}})$. Since decomposition methods search directly for a factorized representation, we inroduce an *operator* $\mathcal{F}_{\text{dec}}$, which performs rank-R approximation and factorization simultaneously, i.e.

$$\mathcal{F}_{\text{dec}}(\theta) = \mathcal{F}_{\text{fact}}(\hat{\theta}) = (\theta_C, \theta_{\text{out}}, \theta_{\text{in}}), \qquad (10)$$

where $\theta_{\text{out}} \in \mathbb{R}^{C_{\text{out}} \times R_{\text{out}}}$, $\theta_{\text{in}} \in \mathbb{R}^{C_{\text{in}} \times R_{\text{in}}}$ are factor matrices, and $\theta_C \in \mathbb{R}^{d \times d \times R_{\text{out}} \times R_{\text{in}}}$ is a core tensor.

An output tensor $Y \in \mathbb{R}^{H' \times W' \times C_{\text{out}}}$ given a layer input $X \in \mathbb{R}^{H \times W \times C_{\text{in}}}$ can be calculated in a consecutive way as follows [13]

$$Z_1 = \theta_{\text{in}} * X, \quad Z_2 = \theta_C * Z_1, \quad Y = \theta_{\text{out}} * Z_2, \quad (11)$$

where operation $*$ denotes a convolution overall common dimensions.

Therefore, the initial convolutional layer with kernel θ can be replaced by three convolutional layers (Figure 3). Indeed, we obtain Z_1, Z_2, Y by sequentially propagating X through the layers with convolutions of spacial sizes 1×1, $d \times d$ and 1×1 respectively. Thus, for the decomposed layer, we get $O(C_{\text{in}}R_{\text{in}} + d^2 R_{\text{out}}R_{\text{in}} + C_{\text{out}}R_{\text{out}})$ layer parameters, and propagation through this layer requires $O(HWC_{\text{in}}R_{\text{in}} + H'W'(d^2 R_{\text{out}}R_{\text{in}} + C_{\text{out}}R_{\text{out}}))$ operations.

---

[3]The dimension-n unfolding of an N-dimensional tensor of size $d_1 \times \ldots \times d_N$ reorders the elements of the tensor into a matrix with d_n rows and $d_1 \ldots d_{n-1}d_{n+1} \ldots d_N$ columns
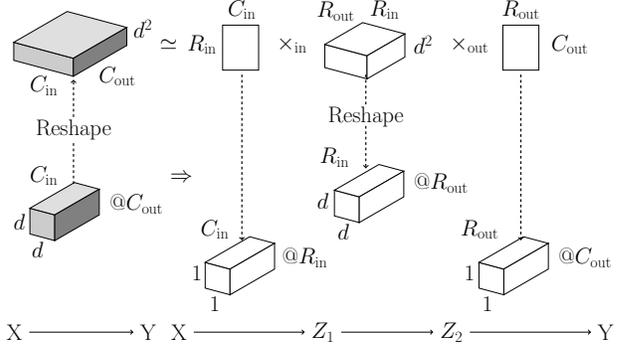

Figure 3. Decomposition of a convolutional layer into three new ones using Tucker-2 kernel approximation. The top row shows an approximation of a 3D weight tensor with a low-rank tensor, which can be represented in Tucker-2 format ($\times_{\text{in}}, \times_{\text{out}}$ denote multilinear products along channel dimensions). The bottom row depicts how the initial layer is replaced with a sequence of 3 convolutional layers. Weights of new layers are the reshaped components of the factorized format for the low-rank tensor. The notation $@C_{\text{out}}$ means that the 4D weight tensor has C_{out} output channels.

4.2. Further compression of a Tucker-2 decomposed layer

To perform further compression, we need to update weights $\theta_{\text{fact}} = (\theta_C, \theta_{\text{out}}, \theta_{\text{in}})$ of the decomposed layer, i.e. to find $\hat{\theta}'_{\text{fact}} = (\theta'_C, \theta'_{\text{out}}, \theta'_{\text{in}})$ such that $\mathcal{F}_{\text{full}}(\hat{\theta}'_{\text{fact}})$ has multilinear rank $R' = (R'_{\text{out}}, R'_{\text{in}})$, $R' \le R$ (elementwise comparison).

The naive way to do that is to obtain a new core and factor matrices by approximating tensor $\mathcal{F}_{\text{full}}(\theta_{\text{fact}})$ with Tucker-2 decomposition (the path along dashed arrows in Figure 2).

We propose to use a more efficient update based on the properties of Tucker decomposition (Figure 4). Namely, we approximate the core θ_C using Tucker-2 and then update the weights in the following way:

$$\mathcal{F}_{\text{dec}}(\theta_C) = (\theta_C^*, \theta_{\text{out}}^*, \theta_{\text{in}}^*),$$
$$\theta'_C = \theta_C^*, \quad \theta'_{\text{in}} = \theta_{\text{in}}\theta_{\text{in}}^*, \quad \theta'_{\text{out}} = \theta_{\text{out}}\theta_{\text{out}}^*. \qquad (12)$$

5. Automated rank selection

In Section 2 we have shown that iterative architecture compression and model fine-tuning (4) is equivalent to iterative parameter set reduction ($\Theta \supset \Theta^{R_1} \supset \Theta^{R_2} \supset \ldots$) for the initial architecture f and fine-tuning with parameter constraints.

To automatically choose gradually decreasing ranks $R_1 > R_2 > \ldots$ for low-rank approximation of θ, we experiment with two different scenarios: Bayesian approach and constant compression rate. Bayesian approach based on rank estimation to remove redundancy from the weight tensor. Constant compression rate approach based on a rank
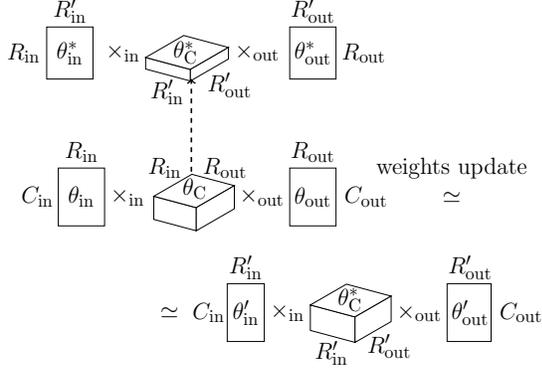
Figure 4. A low-rank approximation of a 3D tensor represented in Tucker-2 format.

calculation to obtain the desired parameter reduction rate after several iterations.

5.1. Bayesian approach

For convenience, we introduce two notations: an *extreme rank* and a *weakened rank*. The extreme rank is the value at which almost all redundancy is eliminated from the tensor after decomposition. The weakened rank is the value at which a certain amount of redundancy is preserved in the tensor after decomposition.

In Bayesian approach, firstly, extreme rank R_e is found via *GAS of EVBMF* (Global Analytic Solution of Empirical Variational Bayesian Matrix Factorization [19]), and secondly, a rank weakening is performed.

The GAS of EVBMF can automatically find matrix rank by performing Bayesian inference, however, it provides a suboptimal solution. Unlike authors of [13], we use GAS of EVBMF not to set a rank for a weight approximation (i.e. $R = R_{\text{evbmf}}$), but only to determine extreme rank (i.e. $R_{\text{extr}} = R_{\text{evbmf}}$)[4].

To determine the extreme rank for Tucker-2 approximation using the GAS of EVBMF, we apply it to the unfoldings of the weight tensor associated with channel dimensions [13]. That is, at the $(k+1)$-th iteration we apply it to matrices of sizes $R_{\text{in}}^k \times d^2 R_{\text{out}}^k$ and $R_{\text{out}}^k \times d^2 R_{\text{in}}^k$.

The weakened rank R_{weak} depends linearly on the extreme rank and serves to preserve more redundancy in the decomposed tensor. Setting $R = R_{\text{weak}}$ facilitates fine-tuning and yields a compression step with better accuracy.

The weakened rank is defined as follows:

$$R_{\text{weak}} = R_{\text{init}} - w(R_{\text{init}} - R_{\text{extr}}), \qquad (13)$$

where w, is a hyperparameter called *weakening factor*, $0 < w < 1$, and R_{init} stands for initial rank. This results in $R_{\text{extr}} \leq R_{\text{weak}} \leq R_{\text{init}}$. Our experiments show that the

---

[4]In [13] instead of EVBMF they use non-empirical version of VBMF, which requires manual setting of additional global parameter.

optimal value for w is in the range: $0.5 \leq w \leq 0.9$. If the initial rank is less than 21, our algorithm considers such kernels as small enough and does not compress them.

5.2. Constant compression rate

Ranks for tensor approximations can be chosen based on the parameter reduction rate that we want to achieve at each compression step. By choosing rank in such a way, we can control the speedup of each convolutional layer.

Suppose we want to reduce the number of kernel parameters in initial convolutional layer, $d^2 C_{\text{in}} C_{\text{out}}$ times α. In Tucker-2 case, having $R_{\text{in}} C_{\text{in}} + R_{\text{out}} R_{\text{in}} d^2 + R_{\text{out}} C_{\text{out}}$ parameters in the decomposed layer, and assuming the multilinear rank has the form $(\beta R, R)$, $\beta > 0$, we can derive

$$R \leq \frac{-\frac{C_{\text{in}} + \beta C_{\text{out}}}{\beta d^2} + \sqrt{\frac{(C_{\text{in}} + \beta C_{\text{out}})^2}{\beta^2 d^4} + \frac{4 C_{\text{in}} C_{\text{out}}}{\beta \alpha}}}{2}. \qquad (14)$$

Therefore, to achieve times α parameters reduction using Tucker-2 tensor approximation, we choose ranks according to the inequality (14).

6. Experiments

In this paper, we focus on compressing object detection models. We apply our MUSCO approach on Faster R-CNN [23] with different backbones. To show the applicability of our method to the variety of tasks, we apply it to compress several classifications and other object detection benchmarks, such as ResNet-18, VGG-16, AlexNet, YOLOv2, Tiny YOLOv2. To one iteration of MUSCO based on Tucker-2 decomposition, we further refer as *Tucker2-iter*.

6.1. Object detection

This section demonstrates the effectiveness of MUSCO algorithm in compressing Faster R-CNN with VGG-16 backbone[5], Faster R-CNN C4 and Faster R-CNN FPN with ResNet-50 backbone[6]. In our experiments, we focus on backbone compression since tensor methods can reduce the parameter redundancy, which usually occurs in convolutional layers. The quality of object detection tasks is evaluated using mAP (mean Average Precision) metrics.

6.1.1 Faster R-CNN with VGG-16 backbone

Faster R-CNN with VGG-16 backbone has been fine-tuned and evaluated on Pascal VOC 2007 [4] train and test datasets respectively.

In Table 1 we have provided all recent results on Faster R-CNN (VGG-16) compression and our best-obtained models. Parameter **nx** means that we select ranks for Tucker-2

---

[5]https://github.com/chenyuntc/simple-faster-rcnn-pytorch
[6]https://github.com/facebookresearch/maskrcnn-benchmark

decomposition based on a constant compression rate strategy. For example, MUSCO(nx, 3.16, 2) is a compressed model obtained after two compression steps using $3.16\times$ parameter reduction at each step.

You can see from the table that the iterative approach allows obtaining higher FLOPs reduction than non-iterative one at a similar mAP level. For example, MUSCO(nx, 1.77, 4) and MUSCO(nx, 3.16, 2) outperform Tucker2-iter (nx, 10).

Comparing to the state-of-the-art methods from Table 1, the MUSCO approach demonstrates significant FLOPs reduction while maintaining the mAP on par with other methods. FLOPs are computed in the same way as in [9].

Model	FLOPs	mAP
FASTER R-CNN with VGG-16 @ VOC2007		
[9] baseline	$1\times$	68.7
Channel Prunning [9]	$4\times$	66.9(-1.8)
Accelerating VD [25]	$4\times$	67.8(-0.9)
AutoML Compression [8]	$4\times$	68.8(+0.1)
Used baseline	$1.0\times$	71.1
Tucker2-iter (nx, 10)	$9.67\times$	68.6(-2.5)
MUSCO(nx, 3.16, 2)	**$10.49\times$**	**69.2(-1.9)**
MUSCO(nx, 1.77, 4)	**$13.95\times$**	**68.3(-2.8)**

Table 1. Comparison of Faster R-CNN (with VGG-16 backbone) compressed models on VOC2007 dataset. MUSCO(nx, 3.16, 2) is a compressed model obtained after two compression steps using $3.16\times$ parameter reduction at each step.

6.1.2 Faster R-CNN with ResNet-50 backbone

In Tables 2 and 3 we provide compression results obtained by applying MUSCO with Tucker-2 approximation to Faster R-CNN C4 and Faster R-CNN with ResNet-50 backbone.

Faster R-CNN has been fine-tuned and evaluated on COCO-2014 dataset. Faster R-CNN with ResNet-50 backbone has been fine-tuned on Pascal VOC 2007/2012 train datasets and evaluated on Pascal VOC 2007 test dataset.

On Pascal VOC2007 MUSCO models achive $1.39\times$ and $1.57\times$ FLOPs reduction with mAP increase by 2.0 and 0.4, respectively (Table 2). On COCO2014 dataset different MUSCO models outperform one-step compression (Table 3).

6.2. Classification

To better demonstrate the efficiency of our approach, we provide here the result of compressing ResNet architecture for classification task on ImageNet. As you can see from the Table 4, our approach provides more accurate compression comparing to state-of-the-art methods. Our model is the most accurate while achieving the biggest FLOPs reduction.

Model	FLOPs	mAP
FASTER R-CNN C4 with RESNET-50 @ VOC2007+2012		
Used baseline	$1.0\times$	75.0
Tucker2-iter (nx, 1.4)	$1.17\times$	76.8(+1.8)
MUSCO(nx, 1.4, 2)	**$1.39\times$**	**77.0(+2.0)**
MUSCO(nx, 1.4, 3)	**$1.57\times$**	**75.4(+0.4)**
Tucker2-iter (nx, 3.16)	$1.49\times$	75.0(+0.0)

Table 2. Comparison of Faster R-CNN (with ResNet-50 backbone) compressed models on Pascal VOC2007 dataset.

Model	FLOPs	mAP	mAP.50
FASTER R-CNN FPN with RESNET-50 @ COCO2014			
Original	$1.0\times$	37.7	59.1
Tucker2-iter(vbmf, 0.7)	$1.2\times$	**36.3(-1.4)**	**57.3(-1.8)**
MUSCO(vbmf, 0.7, 2)	**$1.7\times$**	**36.2(-1.5)**	**57.1(-2.0)**
MUSCO(nx, 3, 4)	**$1.8\times$**	**35.4(-2.3)**	**56.2(-2.9)**
Tucker2-iter(vbmf, 0.9)	$2.0\times$	33.8(-3.9)	54.0(-5.1)

Table 3. Comparisom of Faster R-CNN (with ResNet-50 backbone) compressed models on COCO2014 dataset. MUSCO (vbmf, 0.7, 2) corresponds to the two-iteration compression with automatically selected ranks using GAS of EVBMF and rank weakening with weakening factor equals 0.7.

Method	FLOPs	Δ top-1	Δ top-5
RESNET-18 @ ILSVRC12 dataset			
Network Slimming [17] by [5]	1.39	-1.77	-1.29
Low-cost Col. Layers [3]	1.53	-3.65	-2.3
Channel Gating NN [11]	1.61	-1.62	-1.03
Filter Pruning [15] by [9]	1.72	-3.18	-1.85
Discrimination-aware CP [26]	1.89	-2.29	-1.38
FBS [5]	1.98	-2.54	-1.46
MUSCO **(Ours)**	**2.42**	**-0.47**	**-0.30**

Table 4. Comparison of different compressed versions of ResNet-18 on ILSVRC-12 validation dataset. The baseline model which we compressed came from torchvision. Before compression it achieved 69.76% top-1 accuracy and 89.08% top-5 accuracy. These are the baseline metrics we use to calculate Δ accuracy.

6.3. Benchmark models

We additionally consider several benchmark models to demonstrate the difference between our approach and non-repetitive one, we performed compression using the GAS of EVBMF as the rank selector. Results are shown in Table 5.

Comparing the results of the compression ratio and speed up in all tables, we can say that the iterative approach showed itself better for the considered networks. We can also see that the highest speedup is achieved on CPUs as follows from Table 6. Moreover, we can see that speed up for different processor series is almost the same. To achieve these results, fine-tuning has been performed as long as the error rate

Model	MUSCO	Tucker2-iter
AlexNet	-0.81	-4.2
VGG-16	-0.15	-2.8
YOLOv2	-0.19	-3.1
Tiny YOLOv2	-0.10	-2.7

Table 5. Quality drop after iterative compression and one-time compression. For AlexNet and VGG-16 the metric is Δ Top-5 accuracy, for YOLO - Δ mAP

decreased for both iterative approach and one-time compression approach. For AlexNet and VGG we measured Top-5 accuracy, and for YOLOv2 and Tiny YOLO we measured mAP.

Model	Size	CPU1	CPU2	GPU
AlexNet	4.90×	4.73×	4.55×	2.11×
VGG-16	1.51×	3.11×	3.23×	2.41×
YOLOv2	2.13×	2.07×	2.16×	1.62×
Tiny YOLOv2	2.30×	2.35×	2.28×	1.71×

Table 6. Results of iterative low-rank approximation for AlexNet, VGG-16, YOLOv2, and Tiny YOLO. These tests were performed on CPUs of two different series and on a GPU: Intel Core i5-7600K, i7-7700K and NVIDIA GeForce GTX 1080 Ti respectively.

7. Related work

We propose a multi-stage compression of neural network that is based on gradual low-rank tensor approximation. Our approach is complementary to pruning and other compression methods (*e.g.* quantization). Several papers [7, 8] consider iterative channel pruning methods for model compression. Channel pruning reduces the number of parameters in a neural network by decreasing channel dimensionality of weight tensors. In contrast, our compression approach is based on tensor factorization (Tucker, Canonical Poliadyc, SVD). We exploit a low-rank structure of weight tensors and eliminate parameter redundancy by approximating neural network weights with tensors of special factorized format.

Several papers [2, 14] considered approaches based on tensor decompositions for neural network compression. Most of them suffer from a significant accuracy drop when the compression rate is big. As far as we know, none of them proposed to use a multi-stage approach with gradual low-rank approximations to diminish this drawback. The authors of [2] have demonstrated the successful application of singular value decomposition (SVD) to fully connected layers. Further, the authors of [14] found a way to decompose 4-dimensional convolutional kernel tensor by applying canonical polyadic (CP) decomposition. But these approaches are able to compress only one or a couple of layers. Moreover, for each layer the rank is unique and the process of rank

selection has to be performed manually every time.

Various methods based on quantization have been proposed in [16, 18, 21]. The main goal of quantization is to reduce the number of bits required for weight storage. Our approach differs from that one because we compress networks by decomposing tensors and reducing ranks. It is worth mentioning that quantization can be used after applying our method and serve as an additional method of compression. But quantization may require altering a framework and a significant speedup can be achieved only taking into account the peculiarities of the hardware.

There are several experiments on training low precision networks [1, 22]. Their methods allow to use only 2 bits for weight storage but accuracy is much lower than in full precision networks, and it is not a compression algorithm because such networks have to be trained from scratch.

Another way to compress a whole network was introduced in [13]. The approach used in their work is automated. The authors combined two different decompositions to be able to compress both fully connected and convolutional layers. Unlike [14, 2], the authors have found a way to select ranks without any manual search. Ranks are determined by a global analytic solution of variational Bayesian matrix factorization (VBMF) [19]. We found that the global analytic VBMF provides ranks for which it is difficult to restore the initial accuracy by fine-tuning for deep networks. In our algorithm, we use the global analytic EVBMF but to select the extremal ranks, which will be weakened afterward.

CP decomposition which was used by [14] is a special case of Tucker decomposition, where the core tensor is constrained to be superdiagonal. In our approach, we use Tucker-2 decomposition. To compress fully connected layers we adopt SVD as it was proposed in [2].

Our approach is different from these methods because all of them apply decomposition algorithms only one time per layer, and ranks provided by the global analytic VBMF can be considered as upper bounds for compression. Our algorithm is iterative, and decomposition algorithms can be applied multiple times for the same layer. Moreover, we can achieve a higher compression rate because we do not have such a boundary.

8. Conclusion

In this paper, we addressed the problem of compression of deep convolutional neural networks. We proposed a multi-stage compression algorithm MUSCO for neural network compression, which performs gradual redundancy reduction. Our method consists of two repetitive steps: compression and fine-tuning. The compression step includes automatic rank selection and low-rank tensor factorization according to the selected rank. We evaluated our approach on the following deep networks used for object detection: Faster R-CNN with VGG-16 and ResNet-50 backbones, YOLOv2,

Tiny YOLO and classification: ResNet-18, AlexNet, VGG-16. Experimental results show that our iterative approach outperforms non-repetitive ones in the compression ratio providing less accuracy drop.

Our method is designed to compress any neural network architecture with convolutional and fully connected layers using Tucker-2, CP or SVD decomposition with two different strategies of automatic rank selection. As future work, we plan to increase the variety of matrix/tensor decompositions used by MUSCO approach at the compression step. Also, we will investigate the effect of combining our approach with hardware-dependent approaches such as quantization and channel pruning approaches.

9. Acknowledgements

References

[1] C. Baldassi, A. Ingrosso, C. Lucibello, L. Saglietti, and R. Zecchina. Subdominant dense clusters allow for simple learning and high computational performance in neural networks with discrete synapses. *Physical Review Letters*, 115(12):128101, 2015.

[2] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

[3] X. Dong, J. Huang, Y. Yang, and S. Yan. More is less: A more complicated network with less inference complexity. pages 5840–5848, 2017.

[4] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1), 2015.

[5] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and C. Z. Xu. Dynamic channel pruning: Feature boosting and suppression. In *Proceedings of the International Conference on Learning Representations*, 2019.

[6] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an" explanatory" multimodal factor analysis. *UCLA Working Papers in Phonetics, 16*, pages 1–84, 1970.

[7] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*.

[8] Y. He, J. Lin, Z. Liu, H. Wang, L. J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *The European Conference on Computer Vision*, September 2018.

[9] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*.

[10] C. J. Hillar and L. H. Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):45, 2013.

[11] W. Hua, C. De Sa, Z. Zhang, and G.E. Suh. Channel gating neural networks. *arXiv preprint arXiv:1805.12549*, 2018.

[12] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[13] Y. D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. 2016.

[14] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. 2015.

[15] H Li, A. Kadav, I. Durdanovic, H. Samet, and Graf H.P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2017.

[16] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[17] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[18] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[19] S. Nakajima, R. Tomioka, M. Sugiyama, and S. D. Babacan. Perfect dimensionality recovery by variational Bayesian PCA. In *Advances in Neural Information Processing Systems*, pages 971–979, 2012.

[20] I. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317.

[21] E. Park, J. Ahn, and S. Yoo. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *The European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[23] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[24] L. R. Tucker. Implications of factor analysis of three-way matrices for measurement of change. *Problems in measuring change*, 15:122–137, 1963.

[25] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, 2016.

[26] Z. Zhuang, M Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 883–894, 2018.