Accelerating the Low-Rank Decomposed Models

Habib Hajimolahoseini
Huawei Technologies Canada
habib.hajimolahoseini@huawei.com

Walid Ahmed
Huawei Technologies Canada
walid.ahmed1@huawei.com

Shuangyue Wen
Huawei Technologies Canada
austin.wen1@huawei.com

Yang Liu
Huawei Technologies Canada
yang.liu8@huawei.com

Abstract

Tensor decomposition is a mathematically supported technique for data compression. It consists of applying some kind of a Low Rank Decomposition technique on the tensors or matrices in order to reduce the redundancy of the data.

However, it is not a popular technique for compressing the AI models duo to the high number of new layers added to the architecture after decomposition. Although the number of parameters could shrink significantly, it could result in the model be more than twice deeper which could add some latency to the training or inference. In this paper, we present a comprehensive study about how to modify low rank decomposition technique in AI models so that we could benefit from both high accuracy and low memory consumption as well as speeding up the training and inference.

1 Introduction

With the fast evolution of AI processors used for training the Deep Learning models, the community's focus is moving towards larger and larger models with millions of trainable parameters (Ahmed et al., 2023; Hajimolahoseini et al., 2024b,a, 2023, 2021a). Tuning all of these parameters consumes a huge portion of memory with a huge and exponentially growing computational complexity during both training and inference (Dean et al., 2012). For example, ResNet-152, which is one of the most widely used Convolutional Neural Network (CNN), has more than 60 million parameters and over 11 billion FLOPs. He et al. (2016). In real-time applications especially when these models are deployed on the smartphones and other embedding devices, memory consumption and computational complexity can raise lots of issues including memory and battery life. However, studies show that such large AI models may include a lot of redundancy in the data they are keeping in terms of their weight matrices/tensors inside their layer.

1.0.1 Tensor Decomposition

In contrast with most of the model compression/acceleration techniques, LRD has the well established theoretical foundation with a long history in mathematics Jaderberg et al. (2014). The weights of fully connected (FC) layers of deep learning models are 2D matrices while the filters of convolutional layers are 4D tensors. Therefore, an appropriate matrix or tensor decomposition technique can be applied to decompose them into smaller ones. The goal of Low Rank Decomposition (LRD) is to replace the original tensor/matrix with an approximate tensor/matrix that is close enough to it but with more efficiency in calculations Hajimolahoseini et al. (2021b).

Submitted to 4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024). Do not distribute.

Singular Value Decomposition (SVD) is the most popular method for decomposing the 2D matrices into 2 smaller ones Van Loan (1987). In this approach, each FC layer is replaced by 2 consecutive FC layers whose weights are calculated from the original matrix using SVD algorithm. On the other hand, for convolutional layers, a higher order version of SVD e.g. Tucker is applied in order to decompose them into multiple components De Lathauwer et al. (2000a,b). LRD will be explained in more details in the next sections.

Although LRD has lots of benefits, there are some shortcomings that prevent it from being a popular method in deep learning community. LRD is mostly considered as a type of model compression technique which does not help in terms of acceleration. This is because it will add more layers to the model architecture which makes the model deeper and deeper. This could cause latency during both training and inference. In this paper, we present a comprehensive study about how we can improve the LRD methods in order to use them as a type of model acceleration technique as well. What motivates us to work on improving LRD for deep learning models is that it has a lot of benefits which may not be provided by the other techniques including:

- It is applied only once during the training and takes only few seconds for decomposing the deep models. Therefore it will not add latency to the total training time.
- It has a rich mathematical foundation and is not based on heuristics. Therefore it is a more generic technique which can be applied to different types of models.
- It does not need heavy pre-training because it can start from a large pre-trained model to initialize the compressed model. This is in contrast to many other techniques in which we should train the compressed model from scratch using random initialization. Therefore when using LRD, a few fine-tuning steps are enough to recover most of the accuracy drop.
- It has a built-in one-shot knowledge distillation technique because the new weights in the decomposed model are calculated from the original model which transform the knowledge from the original (teacher) to the decomposed (student) model in one single shot

In the following sections, the LRD and the our proposed modification strategies are explained.

2 Accelerating Low Rank Decomposition

Generally speaking, each convolutional layer in deep learning models include a 4D tensor $\mathbf{W} \in \mathrm{IR}^{\mathrm{C \times S \times h \times w}}$ of trainable parameters, where C and S represent the number of input and output channels, respectively while h and w are the spatial dimensions of the kernels. For 1×1 convolutional and fully connected layer, we can consider $h = w = 1$. Therefore, the 4D tensor $\mathbf{W}$ could be represented in 2D space as $\mathbf{W} \in \mathrm{IR}^{\mathrm{C \times S}}$.

In the case of fully connected or 1×1 convolutional layers, the weight matrix $\mathbf{W}$ is decomposed using Singular Value Decomposition (SVD) as follows Hajimolahoseini et al. (2021b):

$$\mathbf{W} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top, \tag{1}$$

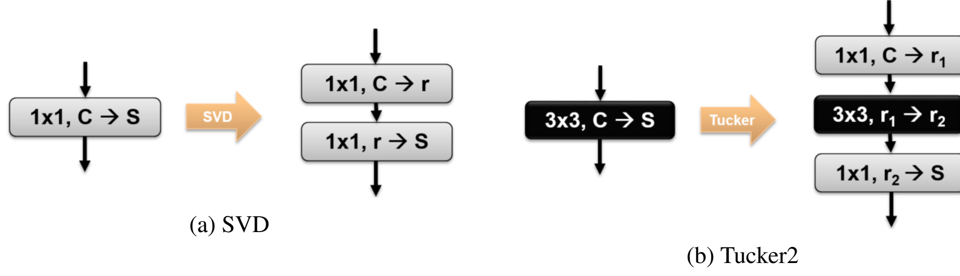
where $\mathbf{U} \in \mathrm{IR}^{\mathrm{C \times C}}$ and $\mathbf{V} \in \mathrm{IR}^{\mathrm{S \times S}}$ are the orthogonal matrices and $\mathbf{\Sigma} \in \mathrm{IR}^{\mathrm{C \times S}}$ is a diagonal rectangular matrix containing the singular values $\sigma_i > 0$ of $\mathbf{W}$ and $r = \min(C, S)$ is called the rank of $\mathbf{W}$ assuming the full-rank.

Using (1) doesn't necessarily lead to compression of the layers. However, if we only use the first $R < r$ components in (1), the resulting matrix is called a low-rank approximation of $\mathbf{W}$:

$$\mathbf{W}' = \sum_{i=1}^{R} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top = \mathbf{U}'\mathbf{\Sigma}'\mathbf{V}'^\top \tag{2}$$

where $\mathbf{U}' \in \mathrm{IR}^{\mathrm{C \times R}}$ and $\mathbf{V}' \in \mathrm{IR}^{\mathrm{S \times R}}$ are the new orthogonal matrices and $\mathbf{\Sigma}' \in \mathrm{IR}^{\mathrm{R \times R}}$ is the new diagonal rectangular matrix. Based on (2), $\mathbf{W}'$ can be interpreted as the multiplication of the following two matrices:

$$\mathbf{W}' = \mathbf{W}_0\mathbf{W}_1, \quad \mathbf{W}_0 = \mathbf{U}'\sqrt{\mathbf{\Sigma}'}, \quad \mathbf{W}_1 = \sqrt{\mathbf{\Sigma}'}\mathbf{V}'^\top \tag{3}$$



(a) SVD

(b) Tucker2

Figure 1: Low Rank Decomposition of 1x1 and 3x3 convolutional layers. Note that FC layers are treated the same as 1x1 Conv layers.

where $\mathbf{W}_0 \in \mathrm{IR}^{\mathrm{C}\times\mathrm{R}}$ and $\mathbf{W}_1 \in \mathrm{IR}^{\mathrm{R}\times\mathrm{S}}$, and $\sqrt{\mathbf{\Sigma}}$ is a diagonal of square root of singular values $\sqrt{\sigma_i}$. According to 3, each fully-connected or 1×1 convolutional layer with weight matrix $\mathbf{W}$ could be replaced with two consecutive layers $\mathbf{W}_0$ and $\mathbf{W}_1$, in which the number of parameters could shrink significantly depending on the low rank R.

In regular convolutial layers, the filters are 4D so a higher order version of SVD e.g. Tucker decomposition is needed. However, in regular CNNs, since spatial dimensions of kernels (h and w) are too small comparing to the feature space dimension (mostly 3 and up to 7 in some models), only the channel related dimensions need to be decomposed. Hence, for simplicity in notations, assuming that $h = w = k$ we can reshape the tensor $\mathbf{W} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{S}\times\mathrm{h}\times\mathrm{w}}$ into a 3D tensor as $\mathbf{W} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{S}\times\mathrm{k}^2}$. Now, the Tucker decomposition can be applied as follow:

$$\mathbf{W} = \mathbf{X} \times_C \mathbf{U} \times_S \mathbf{V} \tag{4}$$

where $\mathbf{U} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{C}}$ and $\mathbf{V} \in \mathrm{IR}^{\mathrm{S}\times\mathrm{S}}$ are unitary matrices and $\mathbf{X} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{S}\times\mathrm{k}^2}$ is the core tensor, containing the 1-mode, 2-mode and 3-mode singular values of $\mathbf{W}$. Symbols $\times_C$ and $\times_S$ also represent multilinear products between each matrix and the core tensor along dimensions C and S, respectively. Simmilar to (1), Tucker decomposition (4) can be rewritten in two steps as follows:

$$\mathbf{Y} = \mathbf{X} \times_C \mathbf{U} = \sum_{i=1}^{r_1} \mathbf{u}_i \mathbf{x}_i^\top \tag{5}$$

$$\mathbf{W} = \mathbf{Y} \times_S \mathbf{V} = \sum_{i=1}^{r_2} \mathbf{y}_i \mathbf{v}_i^\top \tag{6}$$

in which $\mathbf{Y} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{r}_2\times\mathrm{k}^2}$ is the result of multiplying $\mathbf{U}$ with the core tensor $\mathbf{X}$ along dimension C.

For the regular convolutional layers, since $\mathbf{W}$ has a higher number of dimensions, a higher order version of SVD is applied in order to decompose each layer into 3 or more layers De Lathauwer et al. (2000a,b). In this work, we use Tucker decomposition method which replaces each convolutional layer with weight tensor $\mathbf{W} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{S}\times\mathrm{h}\times\mathrm{w}}$ into 3 convolutional layers as follows: a 1x1 convolutional layer with weight matrix $\mathbf{W} \in \mathrm{IR}^{\mathrm{C}\times\mathrm{R}_1}$, followed by a regular convolutional layer called the core with weight tensor $\mathbf{W} \in \mathrm{IR}^{\mathrm{R}_1\times\mathrm{R}_2\times\mathrm{h}\times\mathrm{w}}$, and finally another 1x1 convolutional layer with weight matrix $\mathbf{W} \in \mathrm{IR}^{\mathrm{R}_2\times\mathrm{S}}$ Gusak et al. (2019). R_1 and R_2 are the ranks of Tucker decomposition. The decomposition of 1x1 and 3x3 convolutional layers is illustrated in Fig.1.

Also, the ranks of decomposition can be calculated using different approaches. For more information about different rank selection methods the reader is referred to Gusak et al. (2019); Hajimolahoseini et al. (2021b). Here we calculate the ranks so that each layer has a desired compression ratio.

2.1 Appropriate Rank Selection

The filter dimensions of well-known architectures such as ResNet are selected so that the models could be trained on GPUs in the most efficient way. It could be shown that because of the low level design of the calculations on hardware, some specific dimensions such as powers of 2 would result in a more efficient processing on de devices . That is why all convolutional layers in ResNet models have dimensions that are powers of 2 e.g. 256, 512, etc. However, this is not necessarily the case

after we decompose these models as we calculate the ranks according to a desired compression ratio which may lead to having some odd numbers as the filter dimensions. This may not be efficient in low-level calculations on hardware.

For example, a convolutional layer with filter dimensions of $[512, 512, 3, 3]$ will be decomposed into 3 convolutional layers of dimensions $[512, 309]$, $[309, 309, 3, 3]$ and $[309, 512]$ by applying LRD with 2x compression. Having tensors with dimensions 309 may not lead to efficient calculations on hardware. Therefore, we propose a rank optimization algorithm on top of the LRD's original rank selection method which searches in a domain around the original ranks and finds the candidates that lead to a more efficient calculation for each layer in the model.

The proposed algorithm is explained as a pseudo code in Algorithm 1. As described here, the algorithm starts from the initial rank R (which is calculated according to the desired compression ratio) and decreases it incrementally until it reaches to a the rank which leads to a lower computational time comparing to the original layer. If it could not find such a rank with lower computational time, the original layer will be used instead. This is because for some layers, the original layer may be faster than the decomposed one.

Algorithm 1 Find the rank R_{opt} that leads to more efficient computations

Input: Original layer L, Rank R, Lower bound rank $R_{\min}$, Input tensor x
$T \leftarrow$ Processing time of original layer: $y = L(x)$
Initialization: $r \leftarrow R$ and $R_{opt} \leftarrow 0$
while $r \geq R_{\min}$ **do**
  $L_r \leftarrow$ Decompose layer L using rank r
  $t(r) \leftarrow$ Processing time of decomposed layer: $y = L_r(x)$
  if $r < R$ **then**
    $\Delta t(r) \leftarrow t(r) - t(r-1)$
  end if
  $r \leftarrow r - 1$
end while
Optimal Rank: $R_{opt} \leftarrow \underset{r \in [R_{\min}, R]}{\text{argmax}} \Delta t(r)$
if $t(R_{opt}) < T$ **then**
  Replace L with L_r
else
  Use original layer L
end if

The ranks calculated by the proposed method are reported in Table 1 for the beginning and late layers of ResNet-152 architecture. As seen in this table, some of the layers e.g. layer1.0.conv1 are not decomposed since the original layer is faster compared to the decomposed ones. The other layers are also decomposed using the ranks which result in a faster computation of the output. Note that here we used the PyTorch profiler for calculating the processing time of each layer. In Fig.2, the effect of rank selection on the throughput of layers is depicted. As seen in this figure, changing the Tucker rank from 257 to 256 results in 15% drop in throughput of the layer although the compression ratio stays almost the same (changes less than 1%).

Table 1: Ranks before and after rank optimization algorithm for early and late layers of ResNet-152 on Imagenet dataset

Layer	**# In Channels**	**# Out Channels**	**2x Ranks**	**Optimized Ranks**
layer1.0.conv1	64	64	16	ORG
layer1.0.conv2	64	64	38	32
layer1.0.conv3	64	256	25	24
...				
layer4.2.conv1	2048	512	204	202
layer4.2.conv2	512	512	309	308
layer4.2.conv3	512	2048	204	200
fc	2048	1001	335	253

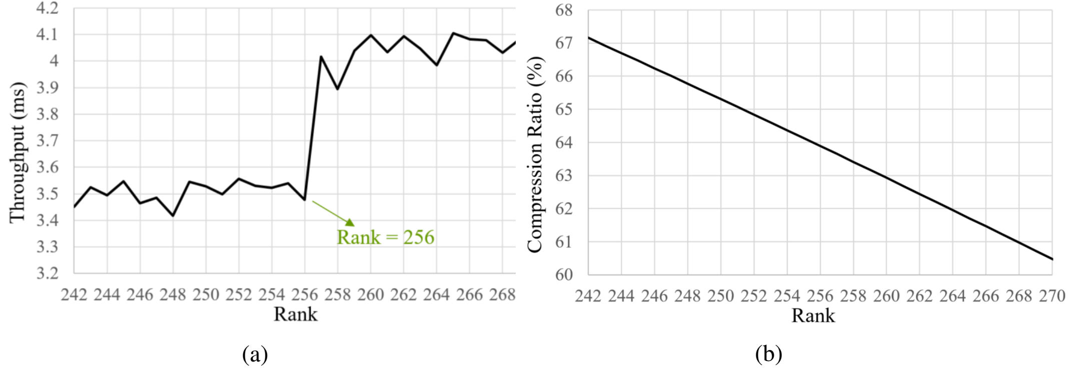


Figure 2: Effect of rank selection on throughput of a 3x3 Conv layer in ResNet-152 with dimensions [512, 512, 3, 3] when decomposed using Tucker2 method with different ranks

2.2 Layer Freezing

Another method we propose for accelerating the decomposed models is to fine-tune only one of the decomposed layers and freeze the rest of them. This is because the decomposed layers are calculated from the original layer using a low rank decomposition algorithm, assuming that decomposed weight tensors are close enough to the original weight tensors when they are reconstructed. Therefore, we can consider the frozen layers as transformation functions and hence, we may not need to update their weights during the optimization. To this end, we freeze the weights of the first 1x1 convolutional layer in Fig.1a and the first and last 1x1 convolutional layer in Fig.1b. This way we can save a lot of time during fine-tuning after decomposing the model. However, note that although this method would accelerate the training phase, the inference speed would be the same as the vanilla LRD method as the number of layers and weights are the same during the inference phase.

2.3 Layer Merging

The previous techniques proposed for accelerating the decomposed models still use the same number of layers as the vanilla LRD. In this section, we propose another approach which can result in a decomposed model with exactly the same number of layers as the original model but with much less number of parameters.

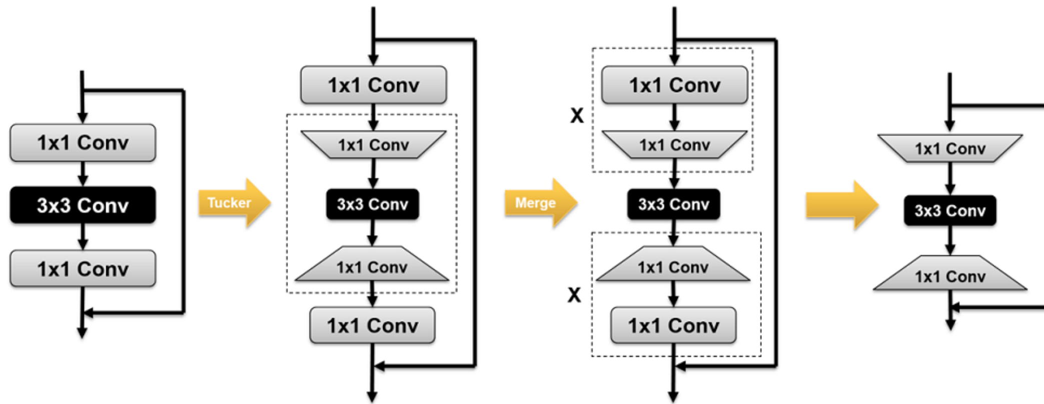


Figure 3: Mixing consecutive 1x1 Conv layers in ResNet modules after applying Tucker decomposition to the middle 3x3 Conv layer.

2.4 Branching Tucker

In Fig.1b, it is shown how Tucker decomposition decomposes each kxk convolutional layer into 3 consecutive conv layers. As seen in this figure, Tucker uses 2 ranks for decomposition r_1 and

r_2. These ranks are calculated so that a desired compression ratio is achieved. For example, for a convolutional layer $\mathbf{W} \in \mathrm{IR}^{\mathrm{C \times S \times h \times w}}$, the ranks could be selected as follow to achieve a compression ratio of α:

$$r_1 = \frac{-\frac{C+\beta S}{\beta k^2} + \sqrt{\frac{(C+\beta S)^2}{\beta^2 k^4} + \frac{4CS}{\beta \alpha}}}{2} \tag{7}$$

However, there is always a trade-off between compression ratio and performance as using smaller ranks for more compression can result in significant drop in accuracy. Therefore, we propose a more efficient way of implementing Tucker decomposition in multiple parallel branches so that with the same large ranks, we can reduce computational cost without compromising the accuracy.

Tucker decomposition shown in (4) can be rewritten in two steps:

$$\mathbf{Y} = \mathbf{X} \times_C \mathbf{U} = \sum_{i=1}^{r_1} \mathbf{u}_i \mathbf{x}_i^\top \tag{8}$$

$$\mathbf{W} = \mathbf{Y} \times_S \mathbf{V} = \sum_{i=1}^{r_2} \mathbf{y}_i \mathbf{v}_i^\top \tag{9}$$

in which $\mathbf{Y} \in \mathrm{IR}^{\mathrm{C \times r_2 \times k^2}}$ is the result of multiplying the first 1x1 conv layer with the core 3x3 conv layer. However, assuming that the ranks r_1 and r_2 are quantized to multiples of integer N:

$$r_1 = NR_1 \tag{10}$$

$$r_2 = NR_2 \tag{11}$$

According to (8) we can write:

$$\mathbf{W} = \sum_{i=1}^{R_2} \mathbf{y}_i \mathbf{v}_i^\top + \sum_{i=R_2+1}^{2R_2} \mathbf{y}_i \mathbf{v}_i^\top + \ldots \sum_{i=(N-1)R_2+1}^{NR_2} \mathbf{y}_i \mathbf{v}_i^\top \tag{12}$$

$$= \sum_{j=1}^{N} \Big(\sum_{i=(j-1)R_2+1}^{jR_2} \mathbf{y}_i \mathbf{v}_i^\top \Big) \tag{13}$$

$$= \sum_{j=1}^{N} (\mathbf{Y}_j \times_{R_2} \mathbf{V}_j) \tag{14}$$

where $\mathbf{Y}_j \in \mathrm{IR}^{\mathrm{C \times R_2 \times k^2}}$ and $\mathbf{V}_j \in \mathrm{IR}^{\mathrm{R_2 \times S}}$ are truncated versions of $\mathbf{Y}$ and $\mathbf{V}$ which include the jth group of R_2 columns of $\mathbf{Y}$ and $\mathbf{V}$, respectively. According to (8), assuming that $\mathbf{X}_j \in \mathrm{IR}^{\mathrm{R_1 \times R_2 \times k^2}}$ and $\mathbf{U}_j \in \mathrm{IR}^{\mathrm{C \times R_1}}$ are the truncated versions of $\mathbf{X}$ and $\mathbf{U}$ we have:

$$\mathbf{Y}_j = \sum_{i=(j-1)R_1+1}^{jR_1} \mathbf{u}_i \mathbf{x}_i^\top \tag{15}$$

$$= \mathbf{X}_j \times_{R_1} \mathbf{U}_j \tag{16}$$

for $j = 1, ..., N$. Substituting (15) into (17) we have:

$$\mathbf{W} = \sum_{j=1}^{N} (\mathbf{X}_j \times_{R_1} \mathbf{U}_j \times_{R_2} \mathbf{V}_j) \tag{17}$$

From this equation, it can be concluded that Tucker decomposition with ranks r_1 and r_2 can be split into N parallel branches, each with smaller ranks of $R_1 = r_1/N$ and $R_2 = r_2/N$. This way, we can reduce computational complexity without even reducing ranks for Tucker decomposition. We can also calculate the weights in each branch from the original weights. This way we don't need to train from scratch.

Fortunately, it can be shown that branched Tucker architecture can efficiently be implemented using grouped convolutions Xie et al. (2017). As depicted in Fig.4, the last two architectures shown in this

figure are equivalent. According to this figure, the total number of parameters in the 3x3 conv layer inside the branched Tucker decomposition would be:

$$= N \times (R_1 \times R_2 \times 9) \tag{18}$$

$$= N \times (\frac{r_1}{N} \times \frac{r_2}{N} \times 9) \tag{19}$$

$$= \frac{1}{N} \times (r_1 \times r_2 \times 9) \tag{20}$$

Comparing to the the total number of parameters in the 3x3 conv layer of the vanilla Tucker i.e. $(r_1 \times r_2 \times 9)$, it means that the layer could be compressed by N times without even reducing the rank.

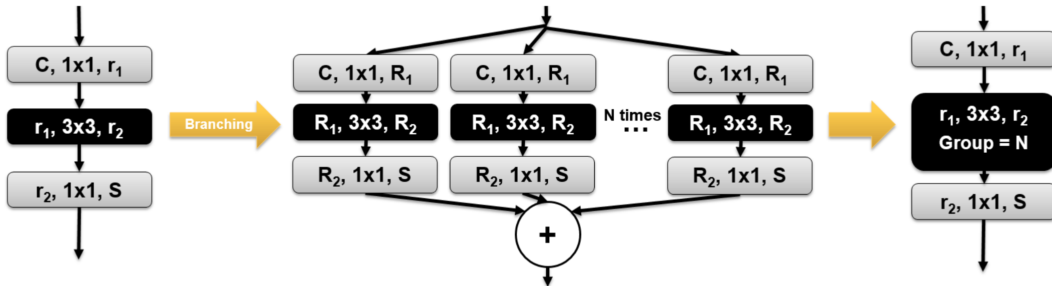


Figure 4: Branched Tucker decomposition and how it can be implemented efficiently using grouped convolutions.

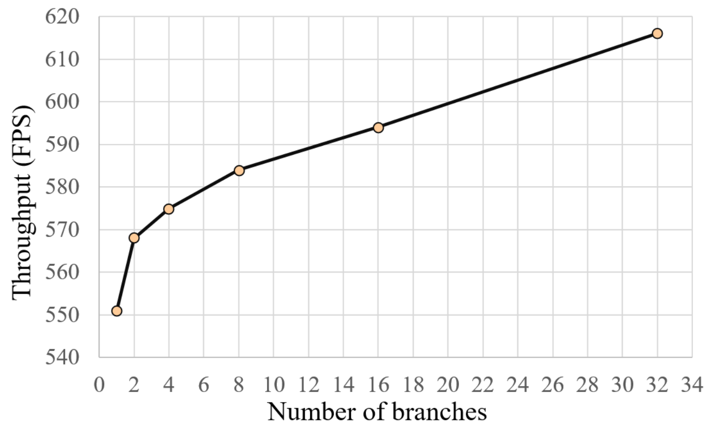


Figure 5: Throughput of the model vs number of branches in each layer for ResNet-152.

3 Experimental Results

Table 2: Statistics of ResNet-50, ResNet-101 and ResNet-152 architectures before and after applying LRD.

Model	Layers	Comp Ratio	ΔFLOPs	Train Speed-up	Infer Speed-up
ResNet-50					
Vanilla LRD	115	-50.00	-43.26	+6.07	+6.82
Optimized Ranks	115	-47.28	-47.71	+14.16	+13.96
Layer Freezing	115	-50.00	-43.26	+24.57	+6.82
Layer Merging	50	-51.49	-55.09	+43.64	+40.58
Layer Branching	0	0	0	0	0
ResNet-101					
Vanilla LRD	233	-50.00	-46.53	+9.66	+10.52
Optimized Ranks	233	-51.96	-49.56	+14.49	+19.92
Layer Freezing	233	-50.00	-46.53	+29.95	+10.52
Layer Merging	101	-56.40	-58.86	+52.66	+54.56
Layer Branching	0	0	0	0	0
ResNet-152					
Vanilla LRD	352	-50.00	-47.69	+11.73	+13.14
Optimized Ranks	352	-51.86	-50.20	+15.86	+18.43
Layer Freezing	352	-50.00	-47.69	+31.72	+13.14
Layer Merging	152	-58.11	-60.18	+55.86	+54.90
Layer Branching	0	0	0	0	0

Table 3: Comparison of accuracy and efficiency for ResNet-50.

Method	Top-1	ΔTop-1	Top-5	ΔTop-5	ΔFLOPs	ΔThroughput
DCP	74.95	-1.06	92.32	-0.61	55.6	-
CCP	75.21	-0.94	92.42	-0.45	54.1	-
MetaPruning	75.40	-1.20	-	-	51.2	-
GBN	75.18	-0.67	92.41	-0.26	55.1	-
HRank	74.98	-1.17	92.33	-0.54	43.8	-
Hinge	74.70	-1.40	-	-	-54.4	-
DSA	74.69	-1.33	92.45	-0.80	-50.0	-
SCP	75.27	-0.62	92.30	-0.68	-54.3	-
LeGR	75.70	-0.40	92.70	-0.20	-42.0	-
NPPM	75.96	-0.19	92.75	-0.12	-56.0	-
Vanilla LRD	76.67	+0.54	-	-	-43.26	+6.82
Optimized Ranks	-	-	-	-	-47.71	+13.96
Layer Freezing	-	-	-	-	-43.26	+6.82
Layer Merging	75.91	-0.21	92.91	+0.04	-55.09	+40.58
Layer Branching	0	0	0	0	0	0

Table 4: Comparison of accuracy and efficiency for ResNet-101.

Method	Top-1	ΔTop-1	Top-5	ΔTop-5	ΔFLOPs	ΔThroughput
Rethinking	75.37	-2.10	-	-	-47.0	-
IE	77.35	-0.02	-	-	-39.8	-
FPGM	77.32	-0.05	93.56	0.00	-41.1	-
NPPM	77.83	+0.46	93.77	+0.21	-56.0	-
Vanilla LRD	76.94	-0.43	93.40	-0.14	-46.53	+13.14
Optimized Ranks	-	-	-	-	-49.56	+18.43
Layer Freezing	-	-	-	-	-46.53	+13.14
Layer Merging	76.55	-0.82	93.4	-0.14	-58.86	+54.90
Layer Branching	76.67	-0.70	93.36	-0.19	0	+7.43

Table 5: Comparison of accuracy and efficiency for ResNet-152.

Method	Top-1	ΔTop-1	Top-5	ΔTop-5	ΔFLOPs	ΔThroughput
Vanilla LRD	-	-	-	-	-47.69	+13.14
Optimized Ranks	-	-	-	-	-50.20	+18.43
Layer Freezing	77.83	-0.48	93.93	-0.11	-47.69	+13.14
Layer Merging	77.86	-0.44	94.116	+0.07	-60.18	+54.90
Layer Branching	77.97	-0.34	93.93	-0.11	-66.75	0

4 Conclusion

In this work, a progressive low rank decomposition method was used for compression of large transformer based language models. In contrast to many of state-of-the-art compression methods where intensive pre-training of the compressed model is necessary, progressive LRD can provide promising performance by compressing the model in the fine-tuning stage. This leads to reduction in the computation resources needed for obtaining a compressed model for a given task. We show that in later steps of the iterative compression where the student models becomes much smaller than the teacher (compression factor larger than $8\times$) KD can be used to improve the performance.

References

Walid Ahmed, Habib Hajimolahoseini, Austin Wen, and Yang Liu. 2023. Speeding up resnet architecture with layers targeted low rank decomposition. arXiv preprint arXiv:2309.12412.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000a. A multilinear singular value decomposition. SIAM journal on Matrix Analysis and Applications, 21(4):1253–1278.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000b. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. SIAM journal on Matrix Analysis and Applications, 21(4):1324–1342.

Jeffrey Dean, Greg S Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, et al. 2012. Large scale distributed deep networks.

Julia Gusak, Maksym Kholiavchenko, Evgeny Ponomarev, Larisa Markeeva, Ivan Oseledets, and Andrzej Cichocki. 2019. Musco: Multi-stage compression of neural networks. arXiv preprint arXiv:1903.09973.

Habib Hajimolahoseini, Walid Ahmed, and Yang Liu. 2024a. Methods, systems, apparatuses, and computer-readable media for decomposing a layer in a neural network. US Patent App. 18/087,877.

Habib Hajimolahoseini, Mohammad Hassanpour, Foozhan Ataiefard, Boxing Chen, and Yang Liu. 2024b. Single parent family: A spectrum of family members from a single pre-trained foundation model. arXiv preprint arXiv:2406.19995.

Habib Hajimolahoseini, Kaushal Kumar, and DENG Gordon. 2023. Methods, systems, and media for computer vision using 2d convolution of 4d video data tensors. US Patent App. 17/502,588.

Habib Hajimolahoseini, Mehdi Rezagholizadeh, Vahid Partovinia, Marzieh Tahaei, Omar Mohamed Awad, and Yang Liu. 2021a. Compressing pre-trained language models using progressive low rank decomposition. Advances in Neural Information Processing Systems.

Habib Hajimolahoseini, Mehdi Rezagholizadeh, Vahid Partovinia, Marzieh Tahaei, Omar Mohamed Awad, and Yang Liu. 2021b. Compressing pre-trained language models using progressive low rank decomposition.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866.

Charles Van Loan. 1987. Matrix computations and signal processing. Technical report, Cornell University.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1492–1500.