Supplementary Material for: Progressive Model Compression with Adaptive Rank Selection in Tucker Decomposition

Anonymous

I. Introduction

The supplementary material covers related work on model compression, and the fundamentals of tensor decomposition, and demonstrates how compression techniques can be applied to fully connected (FC) layers. Finally, this paper introduces the classic neural networks and various datasets used.

II. Related Work

A. Model Compression

Since Denil *et al*. [10] demonstrated the existence of redundancy in neural networks, there has been a surge of outstanding research in this area. Notable methods that have emerged include pruning, weight quantization, and low-rank decomposition.

Weight quantization. Weight quantization is a technique that converts the weight parameters in a neural network from high-precision floating-point numbers (e.g., 32-bit) to low-precision representations (e.g., 8-bit, 4-bit, or even 1-bit). It aims to reduce the computational complexity and storage requirements of neural networks, thereby accelerating the inference process and reducing memory consumption.

Weight quantization has led to the development of various methods to efficiently convert high-precision weights into lower-precision formats. For example, Binary Neural Networks [11], and ABC-Net [12] are proposed to convert floating-point weights to binary values. Jin *et al*. [13] utilized adaptive weights and activation bit-widths to achieve fast deployment across different budget resource platforms. Additionally, a switchable shear level technique is used to further improve the quantization model at the lowest bandwidth. Fei *et al*. [14] employed dynamic programming to achieve optimal bandwidth allocation for weights, and the bandwidth allocation for activation is optimized by considering the signal-to-noise ratio between weights and activation quantization. Furthermore, Huang *et al*. [15] proposed a framework for the formal verification of quantized neural network properties, demonstrating how gradient-based heuristic search methods and boundary propagation techniques can be utilized to improve efficiency. Although these approaches can significantly reduce the storage requirements and computational resources of the model, thus speeding up inference and reducing power consumption, quantization methods usually cause significant accuracy degradation in DNN.

Pruning. Pruning aims to enhance the computational and storage efficiency of neural networks by removing redundant parameters or connections. This process usually involves sparsifying the network weights to reduce model complexity and computational overhead while maintaining predictive accuracy as much as possible.

Early research primarily focused on the importance of weights [16], pruning less important weights based on various criteria. Some approaches attempted to use second-order derivative information to identify unimportant weights [17]. To induce weight sparsity, regularization methods like L1 regularization and dropout were introduced during training. Additionally, regularizing batch normalization parameters was considered to mitigate their adverse effects on trainability. Furthermore, significance computation based on backward graphs is a promising approach; it is a data-free method that calculates significance by estimating the upper bound of reconstruction error in intermediate layers [18]. Recently, pruning schemes based on the lottery hypothesis have gained significant attention. Yin *et al*. [19] monitored and adjusted weight connections in spiking neural networks during the lottery hypothesis phase to ensure optimal utilization of the final lottery when deployed on the hardware. Cunha *et al*. [20] theoretically explored structured pruning from the strong lottery hypothesis and proposed corresponding variants, providing pathways for further research into the hypothesis. Moreover, modern sparse neural network architectures are sensitive to the initial gradient flow; a popular approach is to scale the initial variance of each neuron, which improves gradient flow during early training [21]. However, these methods depend on specific libraries or hardware, and in practice, memory savings are limited since feature maps consume more memory than weights.

Low-Rank Decomposition. Among the various compression techniques, low-rank decomposition stands out as particularly impressive. It decomposes the weight parameters of a neural network into smaller matrices or tensors, effectively compressing the model parameters while maintaining accuracy.

Low-rank decomposition typically involves three stages: pre-training, compression, and fine-tuning. Major compression techniques include TSVD [4], CP [5], Tucker [6], TT [7], and TR [8]. Tensor decomposition methods outperform matrix hierarchies because they extend matrix decomposition techniques to higher orders. This allows them to capture more complex,

higher-order information in neural network parameters. Consequently, methods based on Tucker, TT, and TR are more widely used. Some studies have integrated reconstruction error into the pre-training phase, utilizing the ADMM algorithm to split the original problem into two sub-problems, achieving promising results [22]. However, determining the appropriate rank often relies on enumeration, which is inefficient. Additionally, combining low-rank decomposition with pruning shows significant potential, usually resulting in lower approximation errors [23]. However, the layers that undergo pruning and decomposition require substantial running memory. Encouragingly, Guo *et al*. [24] proposed a low-rank network compression method called low-rank projection with energy transfer, which enhances the performance of low-rank compressed networks by training them from scratch and restoring them to Euclidean space after projection. Nonetheless, these methods still experience varying degrees of accuracy degradation at different compression ratios, making it challenging to balance compression ratio and model accuracy.

Traditional compression methods based on low-rank decomposition often struggle to provide an effective initialization for compressed models, leading to reduced accuracy and lower compression efficiency. Moreover, they are prone to training instability. To tackle these challenges, this paper introduces a progressive compression scheme leveraging Tucker decomposition. In this approach, the pre-trained weights are used in parallel with low-rank adapters, requiring only the fine-tuning of the adapters' parameters. This significantly reduces computational resource requirements and enhances fine-tuning efficiency. Additionally, by employing parallel learning and a gradual parameter decay strategy, the proposed method can achieve better global optimal solutions compared to previous approaches that compressed the model before fine-tuning, resulting in higher accuracy.

B. Rank Selection

In algorithms for model compression based on low-rank decomposition, the choice of rank is pivotal to the compression quality. A low rank tends to severely degrade model accuracy, while a high rank reduces compression performance. The problem of judiciously choosing the rank to balance accuracy with compression ratio is still underexplored. Building on this, some researchers have explored using VBMF [25] to determine the rank in Tucker decomposition. They achieve a layer-wise weight/tensor approximation by minimizing the distance between the pre-trained weights and the compressed weights. Additionally, a reinforcement learning-based tensor rank selection scheme is proposed to enhance the efficiency of deep neural networks through TR decomposition [26]. In contrast, other studies have focused on matrix singular values, setting an energy threshold to truncate unimportant information, thereby ensuring effective compression performance [4]. In above studies, it was assumed that each layer, or even the entire network, had the same rank [7], [8]. However, because different layers exhibit varying levels of information redundancy, imposing a uniform rank hinders the ability to capture redundant features across layers. This, in turn, results in suboptimal compression performance [6].

Departing from prior studies, this paper proposes a dynamic rank selection method for model compression based on Tucker decomposition. We aim to establish the Tucker ranks for different layers and modes, seeking an optimal trade-off between model accuracy and compression ratio.

III. PRELIMINARY

A. Problem Formulation

A deep neural network consisting of L layers needs to sequentially compute feature $\mathbf{Y}^l=\phi(\mathbf{W}^l\mathbf{X}^{l-1}+\boldsymbol{b}^l)$ with Convolutional or FC layers, where $\phi(\cdot)$ is a nonlinear activation function, $\mathbf{X}^l$ and $\mathbf{Y}^l$ represent the input and output feature of lth layer, respectively. To simplify the notation, we use $\{\mathbf{W}^l\}_{l=1}^{\mathbf{L}}$ to denote the set of weights for all layers, and $\mathbf{W}^l$ is a fourth-order tensor and also a learnable parameter. For simplicity, the bias parameters are simplified in this paper. Given a dataset A consisting of B samples with inputs $\mathbf{X}_i$ and outputs $\mathbf{Y}_i$, and a loss function ℓ, the optimization objective function of a standard deep neural network is represented by min $\ell_{(\mathrm{X},\,\mathrm{Y})\in\mathbf{A}}(\mathbf{Y}, f(\mathbf{X};\mathbf{W}))$. Without loss of generality, we aim to compress a convolutional neural network with l layers using a specific parameter-efficient strategy, while carefully balancing the trade-off between model accuracy and complexity.

B. Preliminary

Tucker Decomposition. Tensor decomposition has been widely used in model compression as a generalized low-rank decomposition method. There exists a variety of tensor decomposition methods in tensor theory. Including TK, CP, TT, and TR. TK-2 is chosen as a low-rank decomposition method for compressed models in this paper.

$$\mathbf{V} \approx \mathbf{G} \times_1 \mathrm{A} \times_2 \mathrm{B} \times_3 \mathrm{C} = \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr} a_p \circ b_q \circ c_r \tag{1}$$

$\mathbf{V} \in \mathbb{R}^{I\times J\times K}$ is the input tensor. $\mathrm{A}\in\mathbb{R}^{I\times P}$, $\mathrm{B} \in \mathbb{R}^{J\times Q}$, $\mathrm{C} \in \mathbb{R}^{K\times R}$, and $\mathbf{G} \in \mathbb{R}^{P\times Q\times R}$ represent factor matrix and core tensor.

Tensor Contraction. TK-2 decomposition can be compactly represented by using tensor contraction. Tensor contraction can be executed between any two tensors, provided that they possess at least one dimension in common. For example, suppose two tensors $\mathrm{H} \in \mathbb{R}^{d_1\times d_2\times n}$ and $\mathrm{Q} \in \mathbb{R}^{\mathrm{n}\times d_3\times d_4}$, the output tensor $\mathbf{O}$ of size $\mathbb{R}^{d_1\times d_2\times d_3\times d_4}$ after tensor contraction can be expressed as follows:

$$\mathbf{O}_{(a_1,a_2,b_1,b_2)} = \mathrm{H} \times_1^3 \mathrm{Q} = \sum_{i=1}^{n} \mathrm{H}_{(a_1,a_2,i)} \mathrm{Q}_{(i,b_1,b_2)}. \tag{2}$$

Flatten. The Flatten operation usually refers to the process of converting a multi-dimensional array (or tensor) into a one-dimensional array. This operation is very common in machine learning and data processing, especially when preparing input data or simplifying calculations. For instance, a second-order matrix will be changed from [a_{11}, a_{12}; a_{21}, a_{22}] to [a_{11}, a_{12}, a_{21}, a_{22}] after performing flatten operation.

C. Compressing the FC Layer

For classical neural networks, the output vector $\boldsymbol{y}$ is computed from the input vector $\boldsymbol{x}$ with Eq. 1.

$$\boldsymbol{y}^{\mathrm{T}} = \boldsymbol{x}^{\mathrm{T}}\mathrm{W}, \tag{3}$$

where W is the weight matrix.

To compress the model parameters, this paper employs Tucker-1 decomposition to reduce the weight matrix of the FC layer, which is also referred to as SVD decomposition.

$$\mathrm{W} = \mathrm{USV}^{\mathrm{T}}, \tag{4}$$

here, the TSVD method decomposes the weight matrix into three components: the left singular matrix $\mathrm{U} \in \mathbb{R}^{M\times R}$, the right singular matrix $\mathrm{V}^{\mathrm{T}} \in \mathbb{R}^{R\times N}$, and the diagonal singular value matrix $\mathrm{S} \in \mathbb{R}^{R\times R}$. The parameter R controls the extent to which the information in the matrix is retained.

After decomposition, the low-rank form is easily obtained as follows:

$$\boldsymbol{y}^{\mathrm{T}} = (\boldsymbol{x}^{\mathrm{T}}(\mathrm{US}))\mathrm{V}^{\mathrm{T}}, \tag{5}$$

According to Eq. (5), an FC layer is decomposed into two consecutive sublayers.

$$\boldsymbol{z}^{\mathrm{T}} = \boldsymbol{x}^{\mathrm{T}}(\mathrm{US}), \tag{6}$$

$$\boldsymbol{y}^{\mathrm{T}} = \boldsymbol{z}^{\mathrm{T}}\mathrm{V}^{\mathrm{T}}. \tag{7}$$

where $\boldsymbol{z}$ is intermediate variable.

IV. Experiments and Results

A. Dataset and Basic DNN Model

To get more fair experimental results, we conducted experiments on several widely used DNN models. The detailed information for these architectures is expressed as follows:

VGG-16. VGGNet is a deep neural network architecture that comprises convolutional layers and FC. Variants of the VGG architecture include VGG-13, VGG-16, and VGG-19. The VGG network is widely used in various computer vision tasks, including image classification, object detection, and image segmentation.

ResNet-20, ResNet-32. ResNet-20 and ResNet-32 are variations of the ResNet architecture, designed to address the vanishing gradient problem in DNN through residual learning. ResNet-20 comprises 20 weight layers, while ResNet-32 includes 32 weight layers. Both architectures utilize residual blocks, which enable the network to learn residual mappings rather than direct mappings, thus facilitating the training of deeper models and enhancing performance across various domains such as image classification, object detection, and image segmentation.

We train above models using four different datasets:

MNIST. It is a widely used benchmark dataset in the field of machine learning and computer vision. It consists of 70,000 grayscale images of handwritten digits (0 through 9), each of size 28×28 pixels. The dataset is divided into 60,000 training images and 10,000 test images and is commonly employed for evaluating classification algorithms and models in image recognition tasks.

CIFAR-10. It is a benchmark dataset used in machine learning and computer vision. It contains 60,000 color images across 10 distinct classes, with each class consisting of 6,000 32×32 pixel images. The dataset is split into 50,000 training images and 10,000 test images and is commonly used for evaluating algorithms in image classification and object recognition tasks.

CIFAR-100. It is a benchmark dataset for machine learning and computer vision research. It comprises 60,000 color images, each of size 32x32 pixels, distributed across 100 classes. Each class contains 600 images, with 500 images designated for training and 100 images for testing. The dataset is designed to assess algorithms in fine-grained image classification and object recognition tasks.

SVHN. It is a widely used benchmark in machine learning for digit recognition. It consists of 600,000 color images of house numbers extracted from Google Street View, with each image featuring digits between 0 and 9. The dataset is divided into 73,257 training images, 26,032 test images, and 531,131 extra images for additional training. It is commonly employed to evaluate algorithms for digit classification and object recognition in natural scene imagery. The statistical information of the different datasets is shown in Table I.

TABLE I
STATISTICAL INFORMATION ON DIFFERENT DATA SETS

Datasets	class	Train	Test	Size
MNIST	10	5×10^4	1×10^4	28×28

CIFAR-10	10	5×10^4	1×10^4	32×32
CIFAR-100	100	5×10^4	1×10^4	32×32
SVHN	10	73257	26032	32×32