Facto-CNN: Memory-Efficient CNN Training with Low-rank Tensor Factorization and Lossy Tensor Compression

Seungtae Lee DLTMD1507@G.SKKU.EDU
Jonghwan Ko JHKO@SKKU.EDU
Seokin Hong SEOKIN@SKKU.EDU
Sungkyunkwan University, Suwon, Republic of Korea

Editors: Berrin Yanıkoğlu and Wray Buntine

Abstract

Convolutional neural networks (CNNs) are becoming deeper and wider to achieve higher accuracy and lower loss, significantly expanding the computational resources. Especially, training CNN models extensively consumes memory mainly due to storing intermediate feature maps generated in the forward-propagation for calculating the gradient in the back-propagation. The memory usage of the CNN model training escalates with the increase in batch size and the complexity of the model. Therefore, a lightweight training method is essential, especially when the computational resources are limited. In this paper, we propose a CNN training mechanism called ***Facto-CNN***, leveraging low-rank tensor factorization and lossy tensor compression to reduce the memory usage required in training the CNN models. Facto-CNN factorizes the weight tensors of convolutional and fully-connected layers and then only updates one of the factorized tensors for each layer, dramatically reducing the feature map size stored in the memory. To further reduce memory consumption, Facto-CNN compresses the feature maps with a simple lossy compression technique that exploits the value similarity in the feature maps. Our experimental evaluation demonstrates that Facto-CNN reduces the memory usage for storing the feature maps by 68-93% with a trivial accuracy degradation when training the CNN models.

Keywords: CNN; Training Optimization; Compression; Tensor Factorization

1. Introduction

With the advancements in deep learning, significant progress has been achieved in various fields, such as Computer Vision, Natural Language Processing, and Automatic Speech Recognition. However, these advancements have led to more complex and resource-intensive Deep Neural Network (DNN) models, leading to a high burden for training such models. For instance, a model like GPT-3 (Brown et al. (2020)) has parameter counts of 175 billion, requiring several months for training even with high-performance GPUs (Graphics Processing Units). Similarly, training the state-of-the-art Convolutional Neural Networks (CNNs) models require high computational resources as the number of hidden layers in CNNs increase (He et al. (2016)) and the high-resolution input images are used for training (Karras et al. (2019)).

CNN training typically demands high memory capacity. In CNN training, most memory usage is attributed to feature maps represented as tensors that store activations during the forward pass and are utilized for gradient computation in the backward pass (Rhu et al.

(2016)). The memory usage of the feature maps in the CNN training is at its maximum at the end of the forward pass and gradually decreases during the backward pass. Therefore, reducing the size of feature maps stored in memory during the forward pass becomes crucial to minimize memory usage in CNN training. Furthermore, training with a large batch size significantly escalates memory consumption mainly due to the enlarged memory footprint of the feature maps (Rhu et al. (2016); Jin and Hong (2019)). This high memory requirement of CNN training can limit the use of desirable CNN architectures (e.g., a large number of layers) and the training with a large batch size. This limitation becomes even more pronounced in self-supervised learning (SSL), where large batch sizes are typically required (Chen et al. (2022)). For example, recent SSL methodologies such as SimCLR (Chen et al. (2020)) and BYOL (Grill et al. (2020)) typically use large batch sizes ranging from approximately 1024 to 8192, consequently necessitating a memory-efficient training method.

While CNN training needs high-capacity memory, the GPU systems, commonly employed for both inference and training of deep learning models, have a challenge in meeting this requirement (Rhu et al. (2016); Jin and Hong (2019)). Even if GPUs provide outstanding performance and high programmability, their memory capacity is much smaller than the CPU systems. Since the GPU operates with separate memory spaces from the CPU, transferring the tensors required for computations to the GPU's memory is necessary. When training models using a single GPU, if the memory usage of that GPU exceeds its capacity, the training process comes to an abrupt halt. To mitigate this challenge, some researchers resort to employing multiple GPU systems. However, this approach often introduces imbalanced GPU memory consumption (Zhou et al. (2023)), undermining the GPUs' ability to perform at their optimal capacity.

To tackle this challenge in CNN training, this paper proposes a novel CNN training mechanism called ***Facto-CNN*** that leverages two schemes to reduce the memory footprint of the feature maps in the CNN training: 1) Low-Rank Training (LRT) and 2) Feature Map Compression (FMC). LRT decomposes the tensor of convolutional and fully-connected layers into direction and magnitude matrices by leveraging the low-rank tensor factorization. After that, it only updates the magnitude matrix. Since the feature map used to calculate the gradient for updating the magnitude matrix is much smaller than those required for updating the original tensor, LRT significantly reduces the memory capacity for storing the feature map. To further reduce the memory requirement, Facto-CNN compresses the feature map with a simple lossy compression technique that exploits the data similarity in the feature map.

Our experimental results demonstrate that Facto-CNN reduces the memory usage of feature maps in the CNN training by 68-93% for popular CNN models while maintaining the inference accuracy. With an optimal configuration that delivers a significant reduction in the memory usage with a trivial accuracy degradation (less than 1%), Fact-CNN achieves 93%, 92%, 76%, 77%, 68% reduction in the memory usage of feature maps for VGG11, VGG16 (Simonyan and Zisserman (2014)), fixup-ResNet18, fixup-ResNet34, and fixup-ResNet50 (Zhang et al. (2019)), respectively.

2. Background and Motivation

2.1. CNN Training

Convolutional Neural Network (CNN) is the most popular network for image analysis in various computer vision applications. To accurately classify input images into their respective labels using a CNN model, it is necessary to train the model with a prepared image set. This training process involves using gradient descent to optimize weights. The training process is divided into two stages: forward propagation and backward propagation.

In forward propagation, each layer receives an input feature map and produces an output feature map that is then passed to the next layer. In this process, the fully-connected layer takes an input vector $x \in \mathbb{R}^I$ and performs a vector-matrix multiplication operation with the weight matrix $W \in \mathbb{R}^{O\times I}$ to generate an output $y \in \mathbb{R}^O$:

$$y = Wx \tag{1}$$

Where O is the number of output nodes, I corresponds to the number of input nodes. The convolutional layer, which is the most crucial layer in CNN, performs convolutional operations with an input feature map $X \in \mathbb{R}^{C_I\times H\times W}$ and a kernel $W \in \mathbb{R}^{C_O\times C_I\times K_H\times K_W}$ to produce an output feature map $Y \in \mathbb{R}^{C_O\times(H-K_H+1)\times(W-K_W+1)}$:

$$Y_{f,x,y} = \sum_{i=1}^{K_h}\sum_{j=1}^{K_w}\sum_{c=1}^{C_I} W_{f,c,i,j}X_{c,i+x-1,j+y-1} = W * X \tag{2}$$

H and W represent the height and width of the image, respectively, while C_I and C_O correspond to the input and output channels, and K_H and K_W denote the kernel's height and width.

Also, tensors are stored during the forward pass and used to compute gradients during the backward pass. The tensors stored in each layer are as follows:

- Convolutional and Fully-connected layers: The input feature map and weights are stored.
- Max Pooling layer: The locations of the selected values in the input feature maps are stored. In the PyTorch framework, when an input feature map is flattened through vectorization, the indices of each element in the output feature map are stored. The input feature map is also stored to determine its shape during the backward pass.
- Activation layer(ReLU): Generally, the output of the activation layers needs to be stored. However, nothing is stored since the activation layer's output is the input (input feature map) of a convolutional layer, a fully connected layer, or a pooling layer.

In Backward propagation, the parameters are updated using the Stochastic Gradient Descent (SGD) method defined as the equation 3).

$$W := W - \eta\frac{\partial L}{\partial W} \tag{3}$$

Where η corresponds to the learning rate. To compute $\partial L/\partial W$, layers with parameters (fully connected layers, convolutional layers) need to perform the following operations:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial X}, \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} \tag{4}$$

Where $\partial L/\partial Y$ represents the gradient passed from the next layer. This gradient is passed to the previous layer by multiplying with the weight, $\partial Y/\partial X$. Furthermore, to update the weight value, $\partial L/\partial Y$ is multiplied by the corresponding input value, $\partial Y/\partial W$. Therefore, in the case of a fully connected layer, the equation can be modified as follows:

$$\frac{\partial L}{\partial X} = W^T\frac{\partial L}{\partial Y}, \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}X^T \tag{5}$$

Similarly, convolutional layers can also be expressed as matrix multiplication through Im2col (Chellapilla et al. (2006)), making it possible to compute the gradients of the input and weights similar to fully connected layers.

In the case of ReLU and max pooling, which do not have parameters, only the gradients of the inputs need to be computed and passed back to the previous layer. The max pooling layer utilizes the indices information of the maximum value to propagate gradients to the corresponding location of the maximum value in the input image. Similarly, the ReLU layer propagates gradients to elements in the input image greater than 0.

2.2. Memory Usage in CNN Training

In this subsection, we provide a detailed analysis of the maximum memory usage for storing tensors during training. The memory usage of a CNN can be broadly divided into three parts: feature map, weight, and optimizer state. First, during the forward pass, all layers must store feature maps corresponding to the input or output and the weight values for gradient computation. Table 1 provides the tensors' sizes stored during the forward pass in PyTorch. In the case of fully connected layers, in most scenarios, the memory occupied by weights is larger, as $B < O$, here, B is the batch size. Conversely, in convolutional layers, the memory occupied by feature maps is larger since $B \times H \times W$ is much bigger than $C_o \times K_H \times K_W$. Second, upon completion of the backward pass, the current gradients need to be stored for calculating the momentum of the next gradient update, particularly in optimizers like momentum-SGD (Hinton et al. (2012)) or Adam (Kingma and Ba (2014)). For instance, momentum-SGD (eq. 6), the previous gradient values need to be stored (M_t), necessitating the storage of earlier gradients.

$$\begin{aligned} M_{t+1} &= \alpha M_t - \eta\frac{\partial L}{\partial W_t} \\ W_{t+1} &= W_t + M_{t+1} \end{aligned} \tag{6}$$

The size of the momentum values (optimizer state) to be stored after the backward pass is proportional to the number of trainable parameters, which are updated by the gradients. Note that the number of trainable parameters may not equal the number of weight parameters. In typical supervised learning scenarios, the entire set of weight parameters is updated using gradients so the number of trainable parameters matches the number of weight parameters. However, when only a subset of parameters is updated, the number of trainable parameters becomes smaller.

	tensor size (Byte)
Fully connected layer	$(B \times I) \times 4 + (O \times I) \times 4$
Convolutional layer	$(B \times C_I \times H \times W) \times 4 + (C_O \times C_I \times K_H \times K_W) \times 4$
Max pooling layer	$(B \times C_I \times H \times W) \times 4 + (B \times C_I \times \frac{H}{K_H} \times \frac{W}{K_W}) \times 8$
Activation layer(ReLU)	$(B \times C_I \times H \times W) \times 4 \ or \ (B \times I) \times 4$

Table 1: The size of tensors stored during the forward pass. The numbers 4 and 8 represent tensors with data types float32 and int64, respectively.

2.3. Prior Work - Memory Efficient Training

Data offloading(Rhu et al. (2016), Jin and Hong (2019), Rhu et al. (2018)) is a technique commonly employed in deep learning training processes, which involves temporarily transferring tensors stored in the GPU to the CPU. These tensors are later transferred back to the GPU for usage during the backward pass calculations. Preemptively transferring the tensors required for computations from the CPU to the GPU in the backward pass can help conceal the additional time taken. However, determining the optimal timing for preloading onto the GPU is highly challenging, and significant performance degradation is observed when dealing with large batch sizes (Jin and Hong (2019)). Moreover, in memory-constrained embedded systems, offloading becomes ineffective as the overall memory capacity is insufficient. Therefore, minimizing the size of tensors stored during training is crucial.

The LoRA (Hu et al. (2021)) proposes a method for efficient memory utilization when fine-tuning transformer-based models. Pretrained model such as GPT-3 has excellent performance on various natural language processing tasks. However, tremendous GPU memory is required to fine-tune such models on downstream tasks. Due to the large number of trainable parameters, which is 175 billion, the optimizer state requires a significant amount of VRAM capacity. LoRA addresses these challenges by utilizing low-rank factorization training based on the evidence that the language model has a low “intrinsic rank” (Aghajanyan et al. (2020), Li et al. (2018)). W_q, W_v, W_k and W_0 matrices are replaced by the weight matrix $W + AB$, and only A and B matrices are trained. This method reduces the VRAM usage from 1.2TB to 350GB in the case of GPT-3. However, while this approach reduces the number of trainable parameters, it increases the size of feature maps. This is because, in LoRA, matrix A and matrix B need to be updated (eq 7).

$$\begin{aligned} \frac{\partial L}{\partial A} &= \frac{\partial L}{\partial((W+AB)X)} \frac{\partial((W+AB)X)}{\partial(W+AB)} \frac{\partial(W+AB)}{\partial A} = \frac{\partial L}{\partial Y}(BX)^T \\ \frac{\partial L}{\partial B} &= \frac{\partial L}{\partial((W+AB)X)} \frac{\partial((W+AB)X)}{\partial(W+AB)} \frac{\partial(W+AB)}{\partial B} = A^T \frac{\partial L}{\partial Y} X^T \end{aligned} \tag{7}$$

The equation mentioned above requires storing the matrices for input X and BX. As a result, transformer-based models that primarily rely on fully connected layers experience a significant decrease in VRAM usage. However, due to the large size of feature maps in CNNs, using this approach instead leads to an increase in VRAM usage, making it impractical.

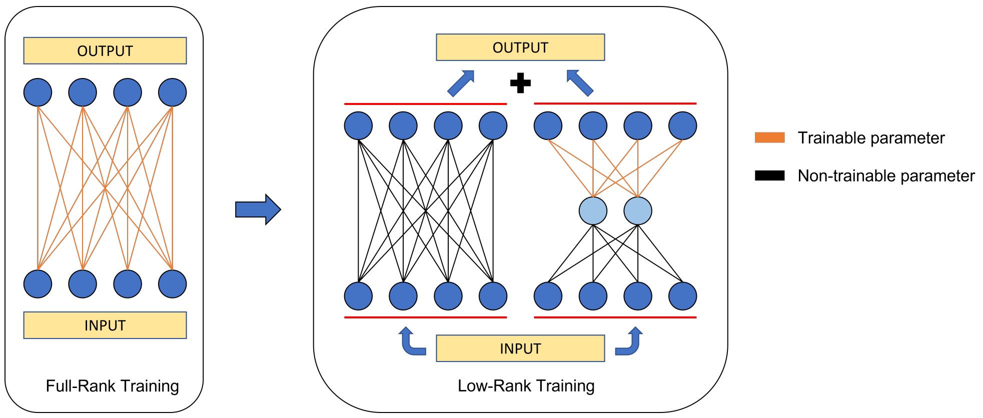


Figure 1: Overview of CNN training with Facto-CNN: The initial few epochs are dedicated to full-rank training, while the remaining epochs utilize low-rank training.

3. Proposed Training Method: Facto-CNN

The overview of the Facto-CNN's training step is illustrated in Figure 1. As shown in this figure, we first train the model for a few epochs, which we refer to as full-rank training (FRT) and is essentially the same as the conventional training method. Afterward, we reconstruct the weight matrix W as $W + MD$ for low-rank training (LRT) and update only the M matrix. Since the rank of M and D is much smaller than that of W, we can observe the following two effects. First, the size of the input feature map stored during the forward pass decrease. Second, the storage required for the optimizer state decreases as the number of trainable parameters decreases. Moreover, lossy compression is employed on the feature maps of the layer that receives input as an image, reducing the height and width dimensions of the feature maps. We will describe Facto-CNN's training mechanism in more detail in the following subsection.

3.1. Low-Rank Training with Tensor Factorization

3.1.1. Layer Reconstruction

To reduce the feature map size stored in memory, Facto-CNN introduces LRT. Facto-CNN begins by reconstructing the weight matrix that has undergone certain epochs with FRT. For fully connected layer's weight matrix $W_{fc} \in \mathbb{R}^{O \times I}$, we add two parameter matrices, namely the direction matrix $D_{fc} \in \mathbb{R}^{R \times I}$ and magnitude matrix $M_{fc} \in \mathbb{R}^{O \times R}$. And for the input vector x to this layer, the output vector y is computed as follows:

$$y = (W_{fc} + M_{fc} D_{fc})x \tag{8}$$

Similarly, for the convolutional layer filter $W_{conv} \in \mathbb{R}^{C_O \times C_I \times K_h \times K_w}$, $M_{conv} \in \mathbb{R}^{C_O \times C_R}$ and $D_{conv} \in \mathbb{R}^{C_R \times C_I K_h K_w}$ matrices are added and the output image Y is computed as follows:

$$\begin{aligned} D_{filter} &= vec^{-1}_{C_R, C_I, K_h, K_w}(vec(D_{conv})) \\ M_{filter} &= vec^{-1}_{C_O, C_R, 1, 1}(vec(M_{conv})) \\ Y &= W_{conv} * X + M_{filter} * (D_{filter} * X) \end{aligned} \tag{9}$$

To reduce computational complexity, operations are performed using the following equation:

$$\begin{gathered}\triangle W_{conv} = vec^{-1}_{C_O,C_I,K_h,K_w}(vec(M_{conv}D_{conv})) \\ Y = (W_{conv} + \triangle W_{conv}) * X\end{gathered} \tag{10}$$

Where R, C_R corresponds to the rank of M, D matrices, which we refer to as "internal nodes" and "internal channels". Moreover, we introduce a vital hyperparameter that determines the internal nodes and channels, called the "internal node ratio(INR)." This hyperparameter takes values between 0 and 1 and is defined as R/I or C_R/C_I.

3.1.2. Matrix initialization

The generated M and D matrices must be appropriately initialized for LRT. (Fig 2) D matrix is randomly initialized using a Gaussian distribution. Then, normalize it along the rows. Normalizing D matrix ensures that all rows have an equal impact on the original weight matrix W. Finally, we multiply α by the D matrix. This α guarantees that the variance of gradients on the full weight matrix ($\Omega = W + MD$) caused by the update of matrix M ($Var[\eta(\partial L/\partial M)D]$) is the same as the variance of gradients in FRT ($Var[\eta(\partial L/\partial W_{FRT})]$). Upon intuitive consideration, when transitioning from FRT to LRT phase, it serves to prevent sudden shifts in gradient variance and enables updates with gradients of a magnitude similar to FRT. To satisfy this condition, the following equation must hold:

$$Var[\eta\frac{\partial L}{\partial W_{FRT}}] = Var[(\eta\frac{\partial L}{\partial M})D] \tag{11}$$

Assume that both FRT and LRT use the same learning rate. $\partial L/\partial M$ is equal to $(\partial L/\partial \Omega)D^T$, and at the onset of LRT, $\partial L/\partial \Omega$ corresponds to $\partial L/\partial W_{FRT}$. This can be applied to the previous equation, leading to the following expression:

$$\begin{gathered}Var[\eta\frac{\partial L}{\partial W_{FRT}}] = Var[(\eta\frac{\partial L}{\partial \Omega}D^T)D] \\ Var[\eta\frac{\partial L}{\partial W_{FRT}}] = I \times Var[\eta\frac{\partial L}{\partial \Omega}]Var[D^TD] \\ \frac{1}{I} = Var[D^TD] \\ \sqrt{\frac{1}{RI}} \simeq Var[D]\end{gathered} \tag{12}$$

Thus, we can set the α value so that the D matrix has the abovementioned variance.

$$\begin{gathered}Var[D] = E[D \circ D] - E[D]^2 = \frac{\alpha^2}{I} - 0 \\ \sqrt{\frac{1}{RI}} = \frac{\alpha^2}{I} \\ \therefore \alpha = \sqrt[4]{\frac{I}{R}}\end{gathered} \tag{13}$$

The symbol $\circ$ denotes element-wise product. By applying this initialization method, all internal nodes have an equal influence on the weight matrix W, generating gradients

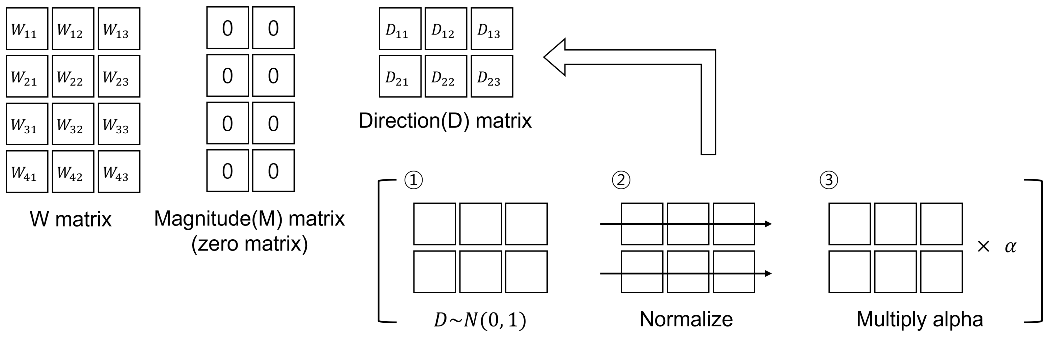
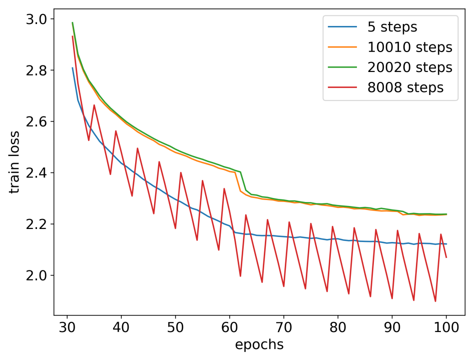
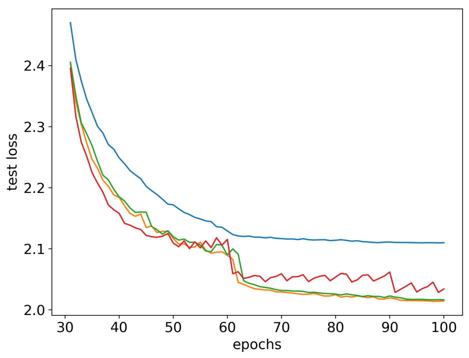


Figure 2: Matrix Initialization for LRT





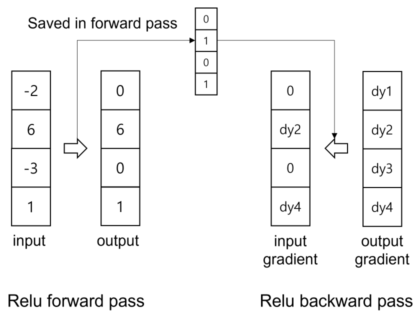
(a) (b)

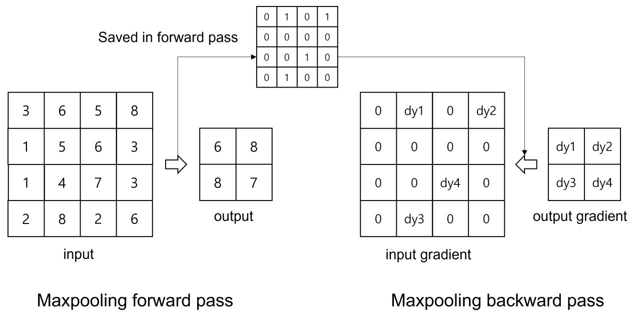
Figure 3: (a) Train loss and (b) test loss when varying the periodicity of adding the MD matrix to W and reconstructing the M and D matrices during training with AlexNet on ImageNet.

of the same variance as those produced in FRT. Additionally, the D matrix consists of R vectors, each with I dimensions, which determine the desired "Direction" for learning. The M matrix determines how much these R vectors should be applied to the W matrix. We only train the M matrix, considering it as the learning of how randomly generated R vectors should be applied to W in terms of their "Magnitude."

3.1.3. Optimizing Magnitude Matrix Update

LRT can lead to underfitting and overfitting for various reasons. Underfitting in LRT is because if the D matrix is not changed during the entire training process, the learning direction does not change, and the weight matrix does not learn in diverse directions, causing the network to saturate quickly. Therefore, adding the MD matrix to W and redefining the M and D matrix at regular training steps is necessary. However, if this process is performed too frequently, overfitting can occur. As shown in Figure 3, if the MD matrix is combined with W with a frequency of less than one epoch (10010 steps), the training loss becomes too high or fluctuates in a zigzag pattern. And if the frequency surpasses one epoch, it results in a higher test loss.





(a) (b)

Figure 4: Feature map binarization for memory saving: (a) ReLU, (b) Max pooling

Another way to prevent overfitting is to apply regularization. We use two regularization methods in LRT. The first method is to apply regularization to matrix M. This adjusts the row vectors of matrix D to be properly applied to the full matrix Ω. The second method is to apply regularization directly to matrix W. Therefore, matrix W is updated at each training step as follows:

$$W := W - \lambda\Omega \tag{14}$$

This allows us to receive similar effects of regularization applied in FRT.

3.2. Auxiliary layer feature map binarization

As layers with parameters no longer store input feature maps, the ReLU layer is required to store the output feature map. However, directly storing the output feature map would consume more memory than FRT. Therefore, to reduce memory usage, we compress the feature maps as the bit-level before storing them (Fig. 4). In ReLU layer, a tensor of the same size as the output is stored. This tensor records a value of 1 if the input is greater than 0 and 0 if the input is less than or equal to 0. During the backward pass, the output gradient and this information are multiplied element-wise to compute the input gradient and pass it back to the previous layer. The same approach can be applied to the max pooling layer as well. The stored tensor contains a value of 1 at the position of the maximum value that was outputted and 0 for the rest of the positions. Then, during the backward pass, the output gradient is upsampled, and the positional tensor is multiplied element-wise to compute the input gradient and pass it backward.

3.3. Lossy Tensor Compression

In addition to the commonly used convolutional and fully connected layers, there are other types of layers in deep learning architectures that also require the storage of feature maps. One such example is a scaling layer that multiplies an input image by a scalar value, which necessitates storing all the input image values solely to update a single parameter. This highly inefficient approach can lead to memory constraints in training CNN models. To address this challenge, we employ mean pooling during the forward pass to compress the data and upsampling during the backward pass to utilize the information. Mean pooling reduces the height and width dimensions of the feature maps, enabling more efficient storage and processing of the data. By averaging the values within each pooling window, the overall spatial resolution is reduced while preserving the essential features of the input.

In the case of convolutional layers, both LRT and feature map compression (FMC) techniques can be applied simultaneously (Figure 5). During the forward pass, the input image undergoes a convolution operation with the Ω filter $(W + \triangle W)$ to generate an output while simultaneously engaging in a convolution operation with the reshaped direction matrix to produce an internal output. This internal output can be seen as the input compressed along the channel dimension, and it is stored in the memory of the GPU after being compressed through lossy compression. Therefore, we can store the feature map compressed for all dimensions except the batch dimension. Subsequently, in the backward pass, the stored tensor is upsampled to restore the internal output. It is then operated on with the output gradient to generate the magnitude matrix gradient. Furthermore, the output gradient is operated on with the omega filter, creating the input gradient. Finally, the input gradient is passed to the previous layer.

3.4. Memory Efficiency

Full rank training requires storing the entire layer's feature map, but if we only train the M matrix parameters, we can store the compressed feature map using the D matrix. In the case of a fully connected layer, we would normally store an $B \times I$ sized matrix. However, to compute the gradient of the M matrix, we only need a tensor of size $B \times R$, where obtained

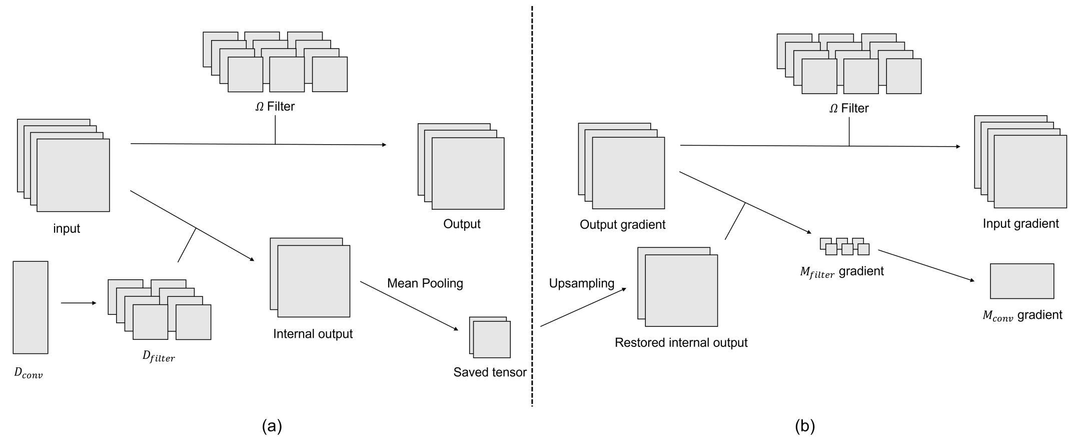


Figure 5: Convolutional layer of Facto-CNN: (a) Forward pass, (b) Backward pass

	Fully-connected layer	Convolutional layer
Input feature map Compression ratio	$1 - R/I$	$1 - C_R/(C_I k_H k_W)$
Trainable parameter Reduction ratio	$1 - R/I$	$1 - C_R/(C_I K_H K_W)$

Table 2: Memory usage reduction with Facto-CNN for fully-connected and convolutional layers

by multiplying the D matrix by the input matrix. Similarly, we only need to store $D_{filter} * X$ for the convolutional layer, which size is $B \times C_R \times H \times W$. Furthermore, lossy compression can be applied, resulting in further compression to the size of $B \times C_R \times (H/k_H) \times (W/k_W)$. Here, k_H and k_W represent the height and width sizes of the lossy compression kernel, respectively. So, the compression ratio is calculated as $1 - R/I$ and $1 - C_R/(C_I k_H k_W)$.

There is also a benefit of reducing the number of trainable parameters. The trainable parameter reduction ratio can be calculated as follows:

$$1 - \frac{M\ matrix\ parameter\ num}{W\ matrix\ parameter\ num} \tag{15}$$

By simple calculation, we can determine that the trainable parameter reduction ratio for the fully connected and convolutional layers are $1-R/I$ and $1-C_R/C_I K_H K_W$, respectively. In particular, the trainable parameter reduction ratio for the convolutional layer is very high. For instance, all convolutional layers of the VGG model employ a 3×3 kernel. When $INR = 0.3$, $K_H = 3$, and $K_W = 3$, the reduction ratio reaches 96.67%. Therefore, only 3.33% of the parameters are used for training.

4. Evaluation

4.1. Methodology

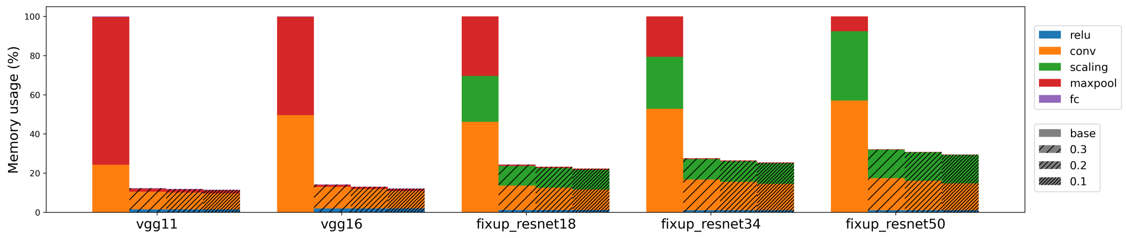
In this section, we compare the accuracy and maximum GPU memory usage of various CNN models (VGG, fixup-ResNet) using the ImageNet dataset(Deng et al. (2009)). ImageNet is an image classification task that contains approximately 1.4 million images and 1000 classes. These images have various sizes, so we cropped them to 224×224 for use. All models were trained for 100 epochs, with the learning rate decreasing by ten at the 30th, 60th, and 90th epochs. FRT was performed for only 30 epochs, and the remaining epochs were trained using LRT. The initial learning rate was set to 0.01 in VGG, while for fixup-resnet, it was set to 0.1. In fixup-resnet, a data augmentation technique mixup (Zhang et al. (2017)) is employed to prevent overfitting. The mixup λ, which determines the interpolation ratio, is set to 0.7 during the FRT step and 0.3 during the LRT step.

4.2. Accuracy and Memory Usage

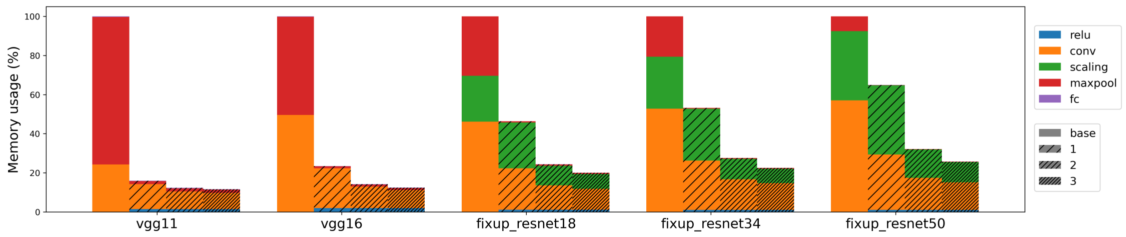
Table 3 presents each model's accuracy and feature map size according to various hyperparameters. For the first line of each model, the feature map size and top-5 accuracy were recorded when applying the general forward and backward functions provided by PyTorch (Paszke et al. (2017)). The hyperparameters used were lossy compression kernel size, INR, and compression apply ratio. Here, The compression apply ratio represents the proportion of convolutional and scaling layers where the feature map reduction technique (LRT + FMC) was applied out of the total number of such layers. We did not use the reduction techniques to the input layer and proceeded to apply compression starting from the layer closest to the input layer in sequential order. In this experiment, VGG models were able to train using only approximately 10% to 20% of the feature map size with no accuracy drop, while Fixup-ResNet models achieved training with an accuracy drop of approximately 1% and using around 20% to 30% of the feature map size.

Model	Lossy compression kernel size	INR	Compression apply ratio	Feature map size	Top-5 accuracy
vgg11(baseline)	-	-	-	11890.5MB (100.00%)	88.620%
vgg11	2	0.3	0.6	1467.3MB (12.34%)	88.480%
	1			1903.7MB (16.01%)	88.708%
	3			1383.3MB (11.63%)	88.592%
		0.4		1515.5MB (12.75%)	88.604%
		0.2		1416.8MB (11.92%)	88.486%
			0.4	2168.6MB (18.24%)	88.696%
			0.8	959.9MB (8.07%)	87.906%
			1.0	793.8MB (6.68%)	87.766%
vgg16(baseline)	-	-	-	17868.5MB (100.00%)	90.382%
vgg16	2	0.3	0.6	2536.1MB (14.34%)	90.402%
	1			4194.4MB (23.47%)	90.460%
	3			2221.2MB (12.43%)	90.401%
		0.4		2717.5MB (15.21%)	90.462%
		0.2		2340.9MB (13.10%)	90.300%
			0.4	3938.7MB (22.04%)	90.608%
			0.8	1678.2MB (9.39%)	89.912%
			1.0	1424.5MB (7.97%)	89.556%
fixup-resnet18(baseline)	-	-	-	3871.5MB (100.00%)	88.412%
fixup-resnet18	2	0.3	0.6	941.6MB (24.32%)	87.534%
	1			1794.4MB (46.35%)	87.760%
	3			774.6MB (20.01%)	87.240%
		0.4		978.3MB (25.27%)	87.632%
		0.2		901.2MB (23.28%)	87.496%
			0.4	1328.1MB (34.30%)	88.324%
			0.8	700.5MB (18.09%)	86.256%
			1.0	577.1MB (14.91%)	85.174%
fixup-resnet34(baseline)	-	-	-	5709.0MB (100.00%)	90.858%
fixup-resnet34	2	0.3	0.6	1575.8MB (27.60%)	89.854%
	1			3040.0MB (53.25%)	90.032%
	3			1287.0MB (22.54%)	89.472%
		0.4		1635.3MB (28.64%)	89.886%
		0.2		1511.1MB (26.47%)	89.686%
			0.4	2158.0MB (37.80%)	90.512%
			0.8	1152.6MB (20.19%)	88.966%
			1.0	837.7MB (14.67%)	87.652%
fixup-resnet50(baseline)	-	-	-	15510.5MB (100.00%)	92.736%
fixup-resnet50	2	0.3	0.6	4985.0MB (32.14%)	91.744%
	1			10066.5MB (64.90%)	92.094%
	3			3983.4MB (25.68%)	91.282%
		0.4		5191.4MB (33.47%)	91.744%
		0.2		4776.5MB (30.89%)	91.658%
			0.4	7080.4MB (45.65%)	92.070%
			0.8	3525.7MB (22.73%)	91.266%
			1.0	2471.0MB (15.93%)	89.940%

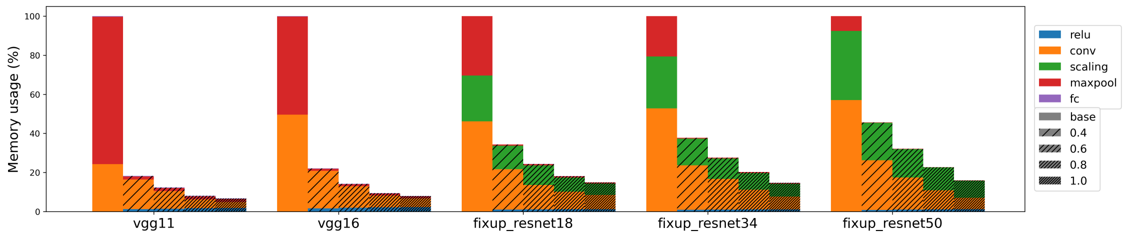
Table 3: Classification accuracy on ImageNet



(a)



(b)



(c)

Figure 6: Memory usage breakdown based on (a) INR, (b) Lossy compression kernel size, and (c) Compression apply ratio.

Figure 6 provides a detailed breakdown of the feature map size saved in each layer, corresponding to the feature map size shown in Table 3. Baseline refers to models using the default models provided by PyTorch. In VGG models, most of the feature map size is occupied by convolutional and max pooling layers. In the case of Fixup-ResNet, scaling layers also contribute significantly. As the figure observed, successful feature map size reduction was achieved across all layers.

5. Discussion

5.1. Training Large CNN Models

Up to this point, we have explored the LRT and FMC techniques that reduce the feature map sizes of layers, including convolutional, fully connected, ReLU, max pooling, and scaling layers. As a result, it is possible to decrease the memory usage of a large CNN model composed of the aforementioned layers. In a theoretical scenario where the lossy compression kernel size, INR, and Compression Application Ratio are set to 2, 0.3, and 0.6, respectively, it is anticipated that the feature map size of fixup-ResNet-101 could be reduced from 23.8GB to 9.0GB, and the feature map size of fixup-ResNet-152 could potentially decrease from 34.1GB to 13.3GB.

5.2. Time Complexity

When employing the Facto-CNN technique for training the model, additional time is introduced beyond the generation of outputs during the forward pass due to the supplementary compression of feature maps. The most time-intensive operation pertains to compressing and storing inputs using LRT. This operation incurs an additional cost of (time taken for output generation * INR). In contrast, only the M matrix necessitates updates during the backward pass, resulting in a reduction of weight matrix update time by a factor of $INR/(K_H K_W)$ times. Consequently, these temporal increments have the potential to offset one another, ultimately resulting in a time complexity akin to that of traditional CNN training. Moreover, if the Ω matrix is precomputed and stored before inference, the inference time aligns with a traditional CNN's.

6. Conclusion and Future works

For CNN training, a significant amount of GPU memory is typically used. This is primarily due to the need to store feature maps of each layer. To address this, we propose compressing the stored information into the input image's channel, width, and height dimensions, excluding the batch dimension. This approach can be applied not only to convolutional layers but also to any layer that performs matrix multiplication operations on weights and inputs. Furthermore, this memory-efficient training method enables on-device training and practical training of large CNN models.

Future work can be considered as follows: 1) In CNNs, the input to the front convolutional layers, which are close to the input layer, have larger height and width dimensions. On the other hand, the back convolutional layers, which are closer to the output layer, have larger channel sizes and relatively more trainable parameters. Therefore, we can explore applying lossy compression to feature maps of front convolutional layers and tensor factorization to back convolutional layers. 2) The inputs and outputs of layers in Graph Neural Network (GNN) and Graph Convolutional Network (GCN) can often contain a significant number of zeros. As a consequence, these matrices tend to possess relatively lower ranks in comparison to dense matrices. This characteristic enables a more favorable application of LRT and FMC techniques. 3) When training a CNN model using SSL, the batch size increases, resulting in larger feature map sizes. Therefore, one possible approach to address this is to consider applying the Facto-CNN method within SSL.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2022R1C1C1012154, 50%), the Ministry of Science and ICT (MSIT) under the Information Technology Research Center (ITRC) support program (IITP-2021-0-02052, 50%) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP). This work was supported by a grant-in-aid of HANWHA SYSTEMS. Seokin Hong is the corresponding author.

References

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

Changyou Chen, Jianyi Zhang, Yi Xu, Liqun Chen, Jiali Duan, Yiran Chen, Son Tran, Belinda Zeng, and Trishul Chilimbi. Why do we need large batch sizes in contrastive learning? a gradient-bias perspective. 2022.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

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

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 835–847, 2019.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4401–4410, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 556–569. IEEE, 2023.