ECAI: Efficient Convolution Activation Inversion for Constant-Memory Convolutional Neural Networks Training

Changhyeon Lee
UNIST, Computer Science and Engineering
changhyeon@unist.ac.kr

Seulki Lee
KAIST, School of Electrical Engineering
seulki.lee@kaist.ac.kr

Abstract

We propose a novel approach that achieves constant activation memory usage during the training of convolutional neural networks (CNNs), addressing a key memory bottleneck in the backward pass. By reconstructing activations required for gradient matrix calculation through the proposed efficient convolution activation inversion (ECAI) rather than storing them in memory during forward pass, it becomes possible to maintain constant activation memory usage across convolution layers. We formulate the activation inversion problem as a set of n systems of linear equations derived from forward convolution operations, and solve them with an accelerated method that achieves $\mathcal{O}(n^2)$ complexity. The proposed approach enables memory-constrained mobile, edge, and embedded devices to perform CNN training without a growth of activation memory over the model capacity while also enhancing training memory efficiency for large-sized images on commercial GPUs. The experimental results demonstrate that the proposed approach maintains constant activation memory by reusing a fixed memory space, improving memory efficiency without degradation in model accuracy. The memory savings achieved by the proposed method increase when using more convolution layers, potentially achieving near-zero activation (e.g., 30× or more activation memory reduction in specific setups). The code implementation is available at an GitHub[1].

[1] `https://github.com/eai-lab/ECAI`

Proceedings of the 29th International Conference on Artificial Intelligence and Statistics (AISTATS) 2026, Tangier, Morocco. PMLR: Volume 300. Copyright 2026 by the author(s).

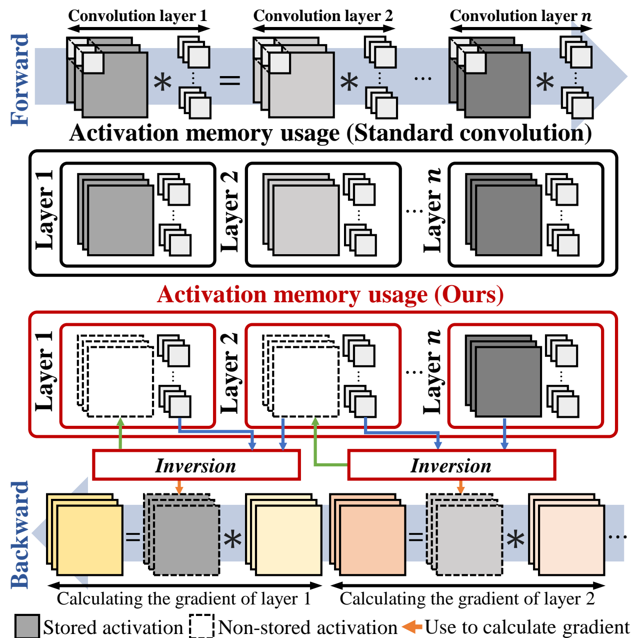

Figure 1: The proposed constant-memory CNN training reconstructs convolution activations of each layer required for backpropagation (Kelley, 1960; Rumelhart et al., 1986) using convolution filters and outputs without storing them in memory, maintaining a fixed activation memory usage in training.

1 INTRODUCTION

Convolutional Neural Networks (CNNs) (Li et al., 2021; Bengio et al., 2017) are among the most widely used deep learning models for a wide range of computer vision tasks, including image classification (Deng et al., 2009), object detection (Lin et al., 2015; Everingham et al., 2010), image and video generation (Ho et al., 2020; Rombach et al., 2022; Ho et al., 2022), and super-resolution (Ronneberger et al., 2015a). While Vision Transformers (Dosovitskiy, 2020; Han et al., 2022) have emerged recently to address more complex tasks, many CNNs, such as ResNet (He et al., 2016)

and YOLO (Redmon, 2016), consistently achieve competitive performance while exhibiting lower complexity and resource requirements. Consequently, CNNs continue to serve as a practical and viable solution not only for a variety of vision tasks but also for applications in other domains, such as speech recognition (LeCun et al., 1995; Abdel-Hamid et al., 2014) and time-series analysis (Zhao et al., 2017a; Luo and Wang, 2024).

Although optimized model architectures (Sandler et al., 2018; Howard et al., 2017) and advanced hardware, such as embedded GPUs (NVIDIA, 2019) and NPUs (Chen et al., 2020), have improved CNN inference efficiency, training CNNs still demand considerable memory resources. This high memory requirement poses a severe obstacle in two real-world scenarios: 1) on resource-constrained devices, where memory limits hinder adaptation techniques like fine-tuning (Tajbakhsh et al., 2016; Zhou et al., 2017) and few-shot learning (Liu et al., 2018a; Satorras and Estrach, 2018), and 2) when processing massive high-resolution data (e.g., $5,000 \times 5,000$ satellite images (Maggiori et al., 2017; Rahnemoonfar et al., 2021)), which often exceeds the memory capacity of even high-end GPUs.

Among the memory components required during training, activation memory typically accounts for the largest portion—constituting up to 97.1% of the total memory (Fig. 2). These activations, i.e., the outputs of each layer during the forward pass, must be stored in memory explicitly for gradient computations during backward pass. Since the volume of activations to be stored in memory grows proportionally with the input image size, the number of CNN layers, and the batch size, the widespread and practical use of CNNs for real-world model adaptation and high-resolution image learning underscores the pressing necessity to limit activation memory growth during CNN training.

To address this challenge, we propose a constant activation memory training method for CNNs, enabled by Efficient Convolution Activation Inversion (ECAI). Unlike the conventional training process, which needs to store activations of each layer in memory for gradient computation, we reconstruct (inverse) the activation of a convolutional layer using the output and the convolution filters (kernels) themselves, as illustrated in Fig. 1. Given that the convolution operation applies multiple filters to the same input feature with overlapping regions, we formulate convolution operations (i.e., matrix multiplications) as a set of linear equations and solve them to reconstruct the activation of the preceding layer. By repeatedly reconstructing (inversing) the necessary activations across convolution layers without accumulating them in memory, it effectively limits the growth of the activation memory usage, enabling constant activation memory training of CNNs regardless of the number of layers.

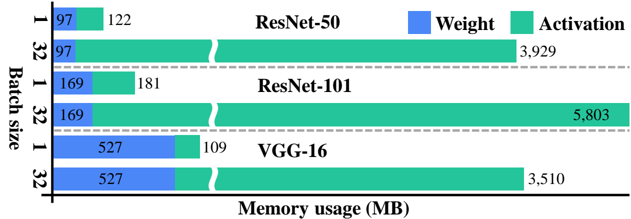

Figure 2: The memory requirements for weight parameters and activations during training of CNN models (i.e., ResNet (He et al., 2015) and VGG (Simonyan and Zisserman, 2015)) on ImageNet (Deng et al., 2009), which are measured with mini-batch sizes of 1 and 32.

To this end, we devise an efficient convolution activation inversion method that solves a set of n systems of linear equations in parallel with a complexity of $\mathcal{O}(n^2)$. This is a significant reduction compared to standard methods such as Gaussian-Jordan Elimination (GJE) (Atkinson, 1991), which typically requires $\mathcal{O}(n^4)$ to solve n systems of linear equations. By parallelizing row-wise operations in GJE and reutilizing a fixed memory space across all convolution layers, n matrix equations can be efficiently solved, while maintaining constant memory usage.

We implement CUDA kernels for the proposed constant-memory CNN training, which is available on an GitHub[1], and deploy them on NVIDIA GPUs (NVIDIA Corporation, 2020) and the Jetson Nano device (NVIDIA, 2019) of 4GB memory. We conduct extensive experiments on various CNN models and tasks, including image classification (He et al., 2015; Simonyan and Zisserman, 2015; Radosavovic et al., 2020; Xie et al., 2017; Sandler et al., 2019), object detection (Ren et al., 2015; He et al., 2018; Khanam and Hussain, 2024), and image segmentation (Long et al., 2015; Chen et al., 2017; Zhao et al., 2017b; Ronneberger et al., 2015b; Liu et al., 2018b; Lin et al., 2017; Khanam and Hussain, 2024), with fine-tuning and full-training. The experimental results show that CNNs can be trained without incurring and storing activations, leading to a significant reduction in activation memory, while maintaining equivalent model performance. Additionally, the proposed method allows for scaling of image sizes and network architecture capacity (i.e., the number of convolution layers) to address larger vision tasks without a growth of activation memory in model training.

2 RELATED WORK

Various methods have been proposed to reduce activation memory during training. Reversible networks (Gomez et al., 2017; Jacobsen et al., 2018; Hascoet et al., 2023) exploit structural invertibility to recover activations during backpropagation, but often rely on strict architectural constraints and introduce additional computational overhead. Rematerialization (Chen et al., 2016; Kumar et al., 2019) and checkpointing (Martens and Sutskever, 2012) avoid storing activations by recomputing them during the backward pass, but incur a high computational cost of $\mathcal{O}(L^2n^3b)$. Compression (Evans and Aamodt, 2021; Liu et al., 2022b, 2021), and approximation (Lee and Lee, 2024; Chakrabarti and Moseley, 2019; Bershatsky et al., 2022) techniques reduce memory by sparsifying or quantizing activations, often at the cost of information loss and degraded performance. In contrast, the proposed method enables exact activation reconstruction with significantly lower overhead $\mathcal{O}(Ln^2b)$, without imposing architectural constraints or sacrificing model accuracy. Details are deferred to the Appendix B.

3 BACKGROUND

3.1 Training of Convolution Layers

Forward Propagation. At the l-th layer of a CNN (Convolutional Neural Network), multiple multiplications are performed between localized regions of the output of the previous $(l-1)$th layer $\mathbf{O}^{l-1} \in \mathbb{R}^{c_i \times h_i \times w_i}$, or equivalently the input $\mathbf{I}^l$, and the filter (kernel) $\mathbf{F}^l \in \mathbb{R}^{c_o \times c_i \times h_f \times w_f}$, which strides over $\mathbf{O}^{l-1}$, to produce the output $\mathbf{O}^l \in \mathbb{R}^{c_o \times h_o \times w_o}$. This feature extraction process, called convolution, is repeated across the convolutional layers as:

$$\mathbf{O}^l = \mathbf{F}^l * \mathbf{O}^{l-1} \text{ where } * \text{ is the convolution operator} \tag{1}$$

During forward propagation, the activation $\mathbf{O}^{l-1}$ must be stored in memory alongside $\mathbf{F}^l$ to compute gradients necessary for updating the weight parameters of $\mathbf{F}^l$ in the subsequent back-propagation process (Kelley, 1960; Rumelhart et al., 1986).

Back Propagation. During the back-propagation, the gradient of the loss, $\mathcal{L}$, with respect to $\mathbf{F}^l$, denoted as $\frac{\partial \mathcal{L}}{\partial \mathbf{F}^l}$, is computed to update the weight parameters of $\mathbf{F}^l$ as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{F}^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{O}^L} \cdots \frac{\partial \mathbf{O}^{l+1}}{\partial \mathbf{O}^l} \frac{\partial \mathbf{O}^l}{\partial \mathbf{F}^l} = \frac{\partial \mathcal{L}}{\partial \mathbf{O}^L} \cdots \mathbf{F}^{l+1} \mathbf{O}^{l-1} \tag{2}$$

Since $\frac{\partial \mathbf{O}^{l+1}}{\partial \mathbf{O}^l} \approx \mathbf{F}^{l+1}$ and $\frac{\partial \mathbf{O}^l}{\partial \mathbf{F}^l} \approx \mathbf{O}^{l-1}$ (from Eq. (1)), the filter $\mathbf{F}^{l+1}$ and the activation $\mathbf{O}^{l-1}$ must be stored in memory for all layers l to compute the gradient as shown in Eq. (2). While the memory requirement for $\mathbf{F}^{l+1}$ remains constant, the memory needed to store the activation $\mathbf{O}^{l-1}$ increases proportionally with the input image size, the number of convolution layers, and batch size. This activation is typically the primary contributor to the overall memory usage in CNN training (Fig. 2).

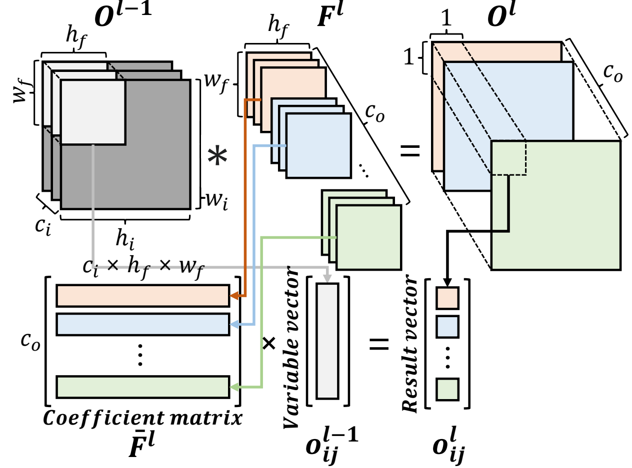

Figure 3: A single matrix equation (bottom) $\bar{\mathbf{F}}^l \mathbf{o}_{ij}^{l-1} = \mathbf{o}_{ij}^l$ is constructed to obtain the activation $\mathbf{O}^{l-1}$ from the convolution operation (upper) using the filter $\mathbf{F}^l$ and output $\mathbf{O}^l$ in Eq. (1).

4 METHOD

To achieve constant activation memory in CNN training, we propose the Efficient Convolution Activation Inversion (ECAI). Unlike the conventional training process, which stores activations of each layer for gradient computation, we reconstruct the convolution activation $\mathbf{O}^{l-1}$ for the l-th layer during backpropagation by solving Eq. (1) for $\mathbf{O}^{l-1}$, given $\mathbf{O}^l$ and $\mathbf{F}^l$. Here, $\mathbf{O}^l$ is the output of the current layer, which we recursively reconstructed from $\mathbf{O}^{l+1}$ and $\mathbf{F}^{l+1}$ at the successive layer. By allocating a fixed memory space $S = \max_{1 \leq l \leq L}(\text{size of } \mathbf{O}^{l-1})$ and reusing it repeatedly across all convolution layers $1 \leq l \leq L$, the activation memory usage effectively remains constant. This contrasts with conventional training process, which requires a cumulative memory of $\sum_{l=1}^{L}(\text{size of } \mathbf{O}^{l-1})$, consequently incurring a substantially increasing memory demand.

4.1 Activation Inversion Formulation

At the l-th convolution layer, the filter $\mathbf{F}^l$ performs a series of matrix multiplications with the activation of the previous $(l-1)$th layer $\mathbf{O}^{l-1}$ (by striding over it,

defined by the stride step, filter size, padding, dilation) to produce the output $\mathbf{O}^l$. We formulate these matrix multiplications into a set of linear equations in the matrix equation form, which can be analytically solved using closed-form methods.

To obtain the activation $\mathbf{O}^{l-1} \in \mathbb{R}^{c_i \times h_i \times w_i}$ in Eq. (1) from $\mathbf{O}^l \in \mathbb{R}^{c_o \times h_o \times w_o}$ and $\mathbf{F}^l \in \mathbb{R}^{c_o \times c_i \times h_f \times w_f}$, we construct a set of matrix equations, as illustrated in Fig. 3. The activation $\mathbf{O}^{l-1}$ and the output $\mathbf{O}^l$ are divided into a set of patches with the row and column index $1 \leq i \leq I$ and $1 \leq j \leq J$ using our patch selection method. The input patch is vectorized into $\mathbf{o}_{ij}^{l-1} \in \mathbb{R}^{n \times 1}$, where $n = c_i \times h_f \times w_f$, and the corresponding output element is vectorized into $\mathbf{o}_{ij}^l \in \mathbb{R}^{c_o \times 1}$. Then, the filter coefficient matrix $\bar{\mathbf{F}}^l \in \mathbb{R}^{c_o \times n}$ is constructed by reshaping the filter tensor $\mathbf{F}^l$, where the k-th row corresponds to the k-th output channel of $\mathbf{F}^l$. By using $\bar{\mathbf{F}}^l$, $\mathbf{o}_{ij}^{l-1}$, and $\mathbf{o}_{ij}^l$, the matrix equation for the ij-th patch $\mathbf{o}_{ij}^{l-1}$ of the activation $\mathbf{O}^{l-1}$ is constructed as $\bar{\mathbf{F}}^l \mathbf{o}_{ij}^{l-1} = \mathbf{o}_{ij}^l$, which is analogous to $\mathbf{Ax} = \mathbf{b}$ in Gauss-Jordan elimination (GJE), as follows:

$$\begin{bmatrix} f_{1,1}^l & \cdots & f_{1,n}^l \\ \vdots & \ddots & \vdots \\ f_{c_o,1}^l & \cdots & f_{c_o,n}^l \end{bmatrix} \begin{bmatrix} o_{ij,1}^{l-1} \\ \vdots \\ o_{ij,n}^{l-1} \end{bmatrix} = \begin{bmatrix} o_{ij,1}^l \\ \vdots \\ o_{ij,c_o}^l \end{bmatrix} \tag{3}$$

By solving Eq. (3) for all $\mathbf{o}_{ij}^{l-1}$, the entire activation $\mathbf{O}^{l-1}$ is reconstructed from $\mathbf{O}^l$ and $\mathbf{F}^l$, allowing its use for the gradient computation in Eq. (2).

4.2 Patch Selection

While it is possible to reconstruct all overlapping patches $\mathbf{o}_{ij}^{l-1}$ through the efficient convolution activation inversion (ECAI), reconstructing $\mathbf{O}^{l-1}$ from all possible $\mathbf{o}_{ij}^l$ is redundant due to the overlaps between $\mathbf{o}_{ij}^l$ made by the striding process of convolution. To eliminate this redundancy by minimizing the number of patches to be inversed, we compute the indices I and J, which select the minimal, non-overlapping set of output patches $\mathbf{o}_{1\leq i \leq I, 1 \leq j \leq J}^l$, required to reconstruct the entire input activation $\mathbf{O}^{l-1}$. The selection of these optimal indices I and J is determined by the activation size, padding, filter size, and stride, as follows:

$$\{I, J\} = \left\{ \frac{h_i - h_p}{\lfloor h_f / h_s \rfloor h_s}, \frac{w_i - w_p}{\lfloor w_f / w_s \rfloor w_s} \right\} \tag{4}$$

where (h_p, w_p) indicate the vertical and horizontal padding, and (h_s, w_s) denote the stride in the vertical and horizontal directions, respectively. Additionally, to ensure complete reconstruction of all regions in $\mathbf{O}^{l-1}$, patches in the lower-right corner that fall outside the calculated striding interval are selected for inversion, according to the following condition:

$$\begin{aligned} I &:= I + 1 \text{ if } \lfloor h_f / h_s \rfloor h_s - h_f > 0 \\ J &:= J + 1 \text{ if } \lfloor w_f / w_s \rfloor w_s - w_f > 0 \end{aligned} \tag{5}$$

4.3 Efficient Activation Inversion

Applying standard GJE to the matrix equation in Eq. (3) incurs $\mathcal{O}(n^3)$ of computation, which grows to $\mathcal{O}(n^4)$ when solving a set of n matrix equations, making it infeasible for applying to practical CNN training. We leverage the CNN property that the same filter $\mathbf{F}^l$ is applied repeatedly across all patches within a layer. This allows us to perform the computationally intensive operation of obtaining the inverse coefficient matrix $(\bar{\mathbf{F}}^l)^{-1}$ only once, and subsequently reuse this matrix to efficiently reconstruct all activation patches. To expedite the entire activation inversion process, we propose an efficient convolution activation inversion (ECAI), which achieves a total complexity of $\mathcal{O}(n^2)$ for both solving the inverse matrix and reconstructing all n activation patches in parallel, as follows.

1) Forward Elimination (Fig. 4.1): As shown in Figure 4, we employ n-parallelization to optimize the row-wise operations inherent in Gaussian-Jordan Elimination (GJE), achieving this by simultaneously updating all column values during each elimination iteration. Further background on these GJE steps is provided in Appendix C. This forward elimination process transforms the filter coefficient matrix $\bar{\mathbf{F}}^l$ into an identity matrix. Furthermore, every operation in this forward elimination process is simultaneously applied to the augmented identity matrix to generate the final inverse coefficient matrix. Within each of the n iterations, the process involves two core steps. Step 1, which is the Normalization of the k-th row based on the diagonal element $(\bar{\mathbf{F}}^l)_{kk}$, requires only 2 operations. Step 2, where Pivoting is applied to the remaining $n-1$ rows, requires just $2(n-1)$ operations per iteration due to the parallelization of row elements by a factor of n. Consequently, the total complexity for forward elimination across all n iterations is calculated as $\sum_{k=1}^{n} 2 + 2(n-1) = \mathcal{O}(n^2)$ operations.

2) Activation Reconstruction via Matrix Product (Fig. 4.2): The inverse coefficient matrix $(\bar{\mathbf{F}}^l)^{-1} \in \mathbb{R}^{n \times n}$ resulting from the forward elimination is then used to reconstruct the entire activation. By concatenating all $\mathbf{o}_{ij}^{l-1}$ vectors into the matrix $\bar{\mathbf{O}}^{l-1} \in \mathbb{R}^{n \times (I \cdot J)}$, the reconstruction is performed by the single matrix product $\bar{\mathbf{O}}^{l-1} = (\bar{\mathbf{F}}^l)^{-1} \bar{\mathbf{O}}^l$, where $\bar{\mathbf{O}}^l$ is the concatenated matrix of patch selected output vectors. The reconstructed inputs $\bar{\mathbf{O}}^{l-1}$ are then reshaped according to each layer configuration to re-

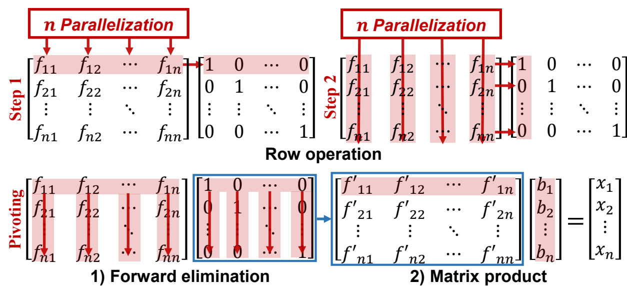

Figure 4: The proposed method ECAI solves n matrix equations in $\mathcal{O}(n^2)$ by leveraging parallel row-wise operations in forward elimination and inner product of the inverse coefficient matrix.

cover their spatial information for use in subsequent gradient computations. Without optimization, this matrix product requires a total complexity of $\mathcal{O}(n^3)$ when $IJ \approx n$. However, by applying n-parallelization to the column computations like forward elimination the complexity of this entire matrix product is reduced from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. This ensures that the time complexity for activation reconstruction remains at a constant level with respect to both the activation size and the batch size, up to the parallelization factor.

Computation Complexity. Consequently, when obtaining patches $\mathbf{o}_{ij}^{l-1}$ for all $1 \leq i \leq I$ and $1 \leq j \leq J$, which is equivalent to solving a set of IJ matrix equations in Eq. (3), the computational complexity of ECAI becomes $\mathcal{O}(n^2)$ when $IJ \approx n$.

5 EXPERIMENT

We implement the proposed constant memory CNN training using CUDA kernels within PyTorch (Paszke et al., 2019), including ECAI, to support deployment on NVIDIA A6000 GPU (48GB) (Corporation, 2020), RTX 3090 GPU (24GB) (NVIDIA Corporation, 2020) and Jetson Nano (4GB) (NVIDIA, 2019). The code implementation is accessible via an GitHub[1], which supports various configurations of convolution operations, such as filter size, stride, padding, and other parameters. Detailed experimental settings are provided in Appendix D.

5.1 Comprehensive Performance Analysis

We begin our experiments by applying the efficient convolution activation inversion to an individual convolution operation to assess its performance, followed by evaluating the training of end-to-end CNN models.

Individual Operations. We begin our experiments by evaluating the proposed ECAI with a single convolution operation over different input image sizes and filter sizes. Table 1 shows the reconstruction error (MSE), the allocated activation memory, and the time required for activation reconstruction. The results demonstrate that activations can be effectively reconstructed with negligible errors (MSE up to 9.4×10^{-10}) and minimal memory allocation; this allocated space is a fixed, reusable constant memory for the entire network. Although the activation inversion process incurs a time overhead, this duration is significantly reduced compared to the standard GJE baseline (Table 6). Furthermore, Figure 5 visually confirms the minimal difference between the original and reconstructed activations, indicating the high fidelity of the reconstruction process.

End-to-End Models. We apply the efficient convolution activation inversion (ECAI) to all convolutional layers of CNN models for various vision tasks and compare their performance and activation memory usage

Table 1: The mean squared error (MSE), memory consumption, and inversion time for a single convolution activation inversion operation are evaluated across varying input and filter sizes—with a fixed stride of one, zero padding, and a batch size of 16—using a PyTorch extension CUDA kernel on an NVIDIA GeForce RTX 3090 GPU.

Input sizes	Filter sizes	MSE	Memory	Time (ms)
[64×64×3]	[3×3, 27]	1.4×10^{-12}	4.0 MB	2.29
[224×224×3]	[3×3, 27]	3.5×10^{-14}	10.0 MB	26.02
[512×512×3]	[3×3, 27]	1.2×10^{-13}	48.0 MB	130.07
[64×64×3]	[5×5, 75]	1.9×10^{-12}	4.0 MB	6.61
[224×224×3]	[5×5, 75]	2.1×10^{-12}	10.0 MB	37.57
[512×512×3]	[5×5, 75]	3.1×10^{-12}	48.0 MB	158.85
[64×64×3]	[7×7, 147]	1.6×10^{-12}	4.0 MB	20.27
[224×224×3]	[7×7, 147]	2.9×10^{-12}	10.0 MB	45.90
[512×512×3]	[7×7, 147]	1.5×10^{-11}	48.0 MB	186.58
[64×64×3]	[9×9, 243]	1.9×10^{-11}	4.0 MB	52.57
[224×224×3]	[9×9, 243]	9.4×10^{-10}	10.0 MB	78.89
[512×512×3]	[9×9, 243]	8.0×10^{-11}	48.0 MB	208.75

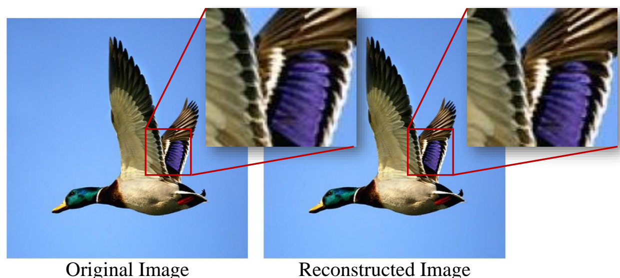

Figure 5: A visual comparison between the original and reconstructed activations (ImageNet (Deng et al., 2009) sample), demonstrating the effectiveness of the proposed convolution activation inversion in addressing numerical challenges during the inversion process.

against CNNs employing standard convolutions. They are trained for image classification (Table 2), object detection (Table 3), and image segmentation (Table 4) tasks using fine-tuning and full-training from scratch. As shown in all models and tasks, the activation memory usage is reduced by an average of 3,113 MB, while achieving equivalent or slightly improved model performance, e.g., improving object detection mAP of Mask R-CNN (He et al., 2018) on VOC-2012 (Everingham et al., 2010) from 0.5086 to 0.5210. We observe a slight improvement in model performance for some tasks, possibly due to random noise introduced by division by small values during row-wise operations in activation inversion for a convolution layer. While such divisions can produce some numerical errors in activations close to zero, which can be propagated through the network, they may, in certain cases, positively influence the learning process. The activation memory reduction achieved by the proposed method becomes more substantial when larger images and an increased number of convolutional layers are employed. For instance, in the case of ResNeXt101 (Xie et al., 2017) on the classification task (Wang et al., 2019) with the batch size of 256, activation memory usage is reduced from 29.3 GB to 12.32 GB. These results demonstrate that ECAI can efficiently reconstruct activations for a series of convolutional layers in a model-wise manner, enabling memory-efficient training process. Although the proposed method is applicable to convolution operations with various configurations, activation memory reduction is maximized when convolution layers utilize the bottleneck structure (Sandler et al., 2019; He et al., 2015; Xie et al., 2017), where output channels are expanded and then reduced using point-wise convolutions. For example, in a comparison between ResNet-50 (He et al., 2015) and ResNeXt-50 (Xie et al., 2017), both with 53 convolutional layers, ResNet-50 utilizing bottleneck blocks, reduces 756 MB activation memory. In contrast, ResNeXt-50, which employs a more aggressive bottleneck structure, reduces 1,406 MB given the same dataset and batch size of 64.

5.2 Memory Usage

Increasing Convolution Layers. Fig. 6 plots the activation memory over the number of convolution layers for different input image sizes on ImageNet (Deng et al., 2009) (224×224) and DeepGlobe2018 (Demir et al., 2018) ($1,024 \times 1,024$), compared with the standard convolution. As shown in the figure, the proposed method maintains constant activation memory (e.g., 85.23 MB and 459.86 MB), even as the number of convolution layers increases, unlike standard convolution, where activation memory increases proportionally with the number of layers (e.g., 2.57 GB and 73.59 GB). When training on extremely large images (GigaPan; NASA) (e.g., $10,000 \times 10,000$), the memory savings achieved through ECAI become more substantial, reducing memory usage from 7.38 TB to 4.55 GB. This not only enables memory-constrained mobile devices to train CNNs within limited memory capacities but also allows high-end systems to learn larger, more complex images with expanded architectural capacity (e.g., convolution layers), which have been previously constrained by the memory limitations of GPUs.

Table 2: The model performance (i.e., f1-score) and activation memory reduction on classification tasks. VGG-16 (Simonyan and Zisserman, 2015) and ResNet-18 are full-trained from scratch on CIFAR-10 (Krizhevsky and Hinton, 2009), while other models are fine-tuned to ImageNet-Sketch (Wang et al., 2019) using pretrained weights from ImageNet.

Image Classification			
Models	f1-score (base)	f1-score (ours)	Memory ⇓
VGG-16	0.9355	0.9323	4.73 MB
ResNet-18	0.9375	0.9395	0.66 MB
ResNet-50	0.8471	0.8529	756.50 MB
RegNet-y-400mf	0.8189	0.8187	276.73 MB
ResNeXt50-32x4d	0.8787	0.8819	1406.00 MB
MobileNet-v2	0.7755	0.7757	316.20 MB

Table 3: The model performance (i.e., mAP) and activation memory reduction on object detection tasks. All models are fine-tuned to VOC-2012 from pretrained weights of COCO (Lin et al., 2015).

Object Detection			
Models	mAP (base)	mAP (ours)	Memory ⇓
Faster R-CNN	0.4938	0.5064	6778.72 MB
Mask R-CNN	0.5086	0.5210	6786.92 MB
YOLO-v5	0.5330	0.5330	5072.64 MB

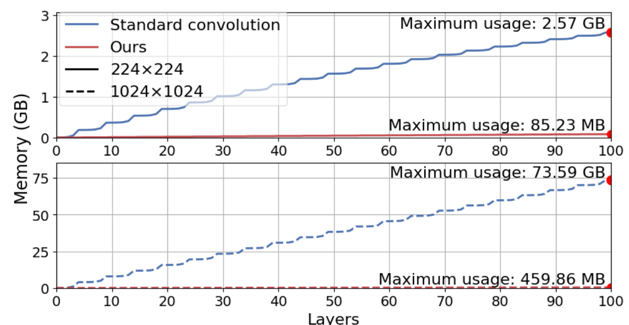

Figure 6: The activation memory growth when convolution layers of the bottleneck module (Sandler et al., 2019; He et al., 2015; Xie et al., 2017) are stacked up to 100 layers, compared against standard convolution; the activation memory stays nearly constant (e.g., 85.23 MB and 459.86 MB), which are 30× and 160× smaller than those of the standard convolution.

Table 4: The model performance (i.e., mIoU) and activation memory reduction on image segmentation tasks. All models are fine-tuned to VOC-2012 (Everingham et al., 2010); FCN (Long et al., 2015) and DeepLab-v3 (Chen et al., 2017) use pretrained weights from COCO dataset, while other models use ImageNet-pretrained backbone models.

Image Segmentation			
Models	mIoU (base)	mIoU (ours)	Memory ⇓
FCN	0.7180	0.7019	5721.96 MB
DeepLab-v3	0.7428	0.7678	7792.71 MB
PSPNet	0.4132	0.4399	1750.39 MB
U-Net	0.6236	0.6355	2540.34 MB
PAN	0.7358	0.7435	3754.57 MB
FPN	0.7276	0.7344	1137.45 MB
YOLO-v5	0.5000	0.5030	5723.16 MB

Partial Activation Storing. The proposed ECAI avoids storing convolution activations in memory, instead reconstructing them at run-time during backward pass. However, when the number of output channels c_o is smaller than $n = c_i \times h_f \times w_f$, i.e., $c_o < n$, the matrix equation in Eq. (3) becomes analytically unsolvable, as the number of equations c_o is smaller than the number of unknowns n. To address this, we selectively store a fraction of activation values in memory to satisfy the condition $c_o \geq n$, which enables the construction of a set of solvable n equations. By doing so, it becomes possible to support a wide range of convolution configurations for activation inversion. Fig. 7 presents the number of activation values stored in memory required for the inversion process in a multi-line graph with different configurations of input channels, output channels, and filter sizes. It shows that the proposed ECAI can be flexibly applied to a wide range of convolution configurations by storing only a fraction of activations in memory as needed.

5.3 Time Measurement

Comparison to Gauss-Jordan Elimination. Table 6 presents a comparison of computation times for solving matrix equations of varying sizes n between ECAI and GJE. The results show that the time required for the inversion process is significantly lower than GJE when solving a single matrix equation (e.g., 249ms vs. 17sec). This disparity becomes even more pronounced when solving multiple matrix equations (e.g., 256ms vs. 157sec). While ECAI takes significantly lower time, we expect that it can be further accelerated by relaxing the strict reconstruction of activations and incorporating activation approximation methods (Lee and Lee, 2024; Chakrabarti and Moseley, 2019; Evans and Aamodt, 2021; Liu et al., 2022a), which we leave for future work.

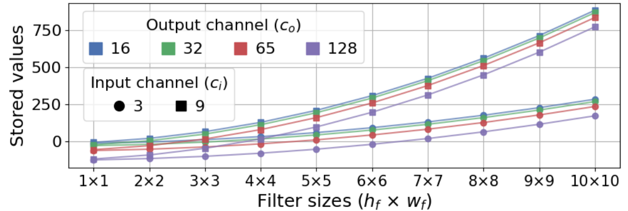

Figure 7: The number of stored activation values as a function of the difference between n and c_o. Different markers represent variations in input channels, and colors denote distinct output channels. Values below zero indicate no activations are stored, whereas values above zero imply the partial storage of activations.

Hardware Dependency. Although CNN training is typically infeasible under extreme memory constraints, the proposed efficient convolution activation inversion (ECAI) enables CNN training in such environments with minimal computational overhead, even on hardware with limited processing cores. Table 5 presents the end-to-end training time of convolution operations on three platforms—NVIDIA A6000, RTX 3090, and Jetson Nano—representing server-grade, general-purpose, and edge-device hardware, respectively. The results in Table 5.(a) indicate that the execution time of ECAI (CUDA kernel) remains consistent across the batch size. This suggests that, as the batch size increases or convolution configurations vary, the proposed method can effectively maintain a trade-off between memory efficiency and computational cost compared to standard convolution. Furthermore, as shown in Table 5.(b), on the edge device (Jetson Nano) with limited GPU performance and memory, the training time for ECAI increases by only 34% for 5×5 kernels. When considering only the time required for the activation inversion for a convolution layer, the time increase is merely 15%, demonstrating the efficiency of the inverse process, particularly in resource-constrained environments. These results show that ECAI is applicable to various multi-threaded platforms, with particular benefits for low-end edge devices (e.g., Jetson Nano). On average, the end-to-end training time increases by 17%, excluding the proposed efficient convolution activation inversion (ECAI) implemented with CUDA kernels. This computational overhead primarily stems from the communication cost associated with the PyTorch Extension (Paszke et al., 2019). This PyTorch-side overhead can be alleviated by integrating the proposed CUDA kernels directly into the PyTorch library, enabling more efficient and seamless execution of ECAI CUDA kernel. In addition, overheads from matrix rearrangement and tensorization, as well as limitations of using only CUDA kernels for optimization, are discussed in Appendix A.

Table 5: The end-to-end training time (in milliseconds), including both forward and backward passes, is compared among standard convolution (Base), the proposed method (Ours), and non-parallelized Gauss-Jordan elimination (GJE). The numbers in parentheses denote the CUDA kernel execution time for activation inversion in a convolution layer. (a) Input size: (3, 64, 64), filter size: 5×5, and batch size $\mathcal{B}$ ranging from 1 to 64. (b) The same input with filter sizes of 3×3 and 5×5, using the batch size of 8.

(a) RTX3090 and A6000 GPUs

	RTX3090			**A6000**		
$\mathcal{B}$	Base	Ours	GJE	Base	Ours	GJE
1	1.43	7.29(5.52)	5,053(5,050)	3.98	9.57(5.06)	4,829(4,821)
8	1.83	7.63(5.73)	44,037(40,757)	6.36	11.32(5.78)	38,665(38,582)
16	2.10	8.43(5.77)	79,500(79,496)	6.80	12.54(5.79)	77,196(77,164)
32	2.86	9.25(5.79)	158,423(158,417)	7.50	13.75(5.80)	154,337(154,326)
64	3.91	10.31(5.78)	329,218(329,212)	7.59	13.53(5.81)	310,865(310,855)

(b) Jetson Nano

NVIDIA Jetson Nano			
Filter size	3×3		
Method	Base	Ours	GJE
Time	26.81	60.82(10.48)	23,566(23,510)
Filter size	5×5		
Method	Base	Ours	GJE
Time	107.49	144.89(16.42)	143,548(143,425)

Table 6: The computation time required to solve matrix equations for the proposed efficient convolution activation inversion (ECAI) in comparison with Gauss-Jordan elimination (GJE). When solving multiple (nine) matrix equations, the proposed ECAI demonstrates only minimal increases in computation time, whereas GJE exhibits up to a 9.5× increase.

(a) Single matrix equation

Matrix size n	32	64	128	256	512
ECAI (ours)	0.70ms	2.76ms	11.92ms	53.17ms	249.41ms
GJE (standard)	2.89ms	21.39ms	250.60ms	2.04sec	17.22sec

(b) Multiple (nine) matrix equations

Matrix size n	32	64	128	256	512
ECAI (ours)	1.48ms	3.38ms	12.82ms	53.51ms	256.80ms
GJE (standard)	28.33ms	186.24ms	2.02sec	19.08sec	157.07sec

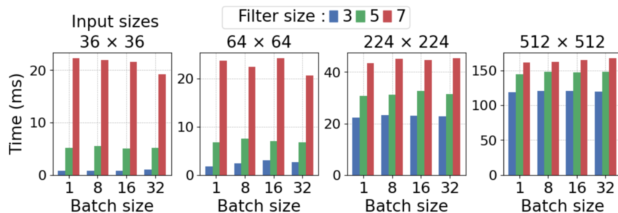

Figure 8: The time required for activation inversion across input sizes of 36, 64, 224, and 512, with batch sizes of 1, 8, 16, and 32. Each color bar represents a different filter size of 3×3, 5×5, and 7×7.

Input Image, Filter, and Batch Size. While the proposed efficient convolution activation inversion (ECAI) takes $\mathcal{O}(n^2)$, it reconstructs all activation patches at once in parallel by performing matrix products of the inverse coefficient matrix. This allows for the activation inversion time to remain nearly independent of the batch sizes until limited by the parallelization factor. Fig. 8 shows the activation inversion time for different configurations of input image, filter, and batch sizes. It is observed that the computation time tends to increase linearly with the input size, while not increasing substantially when batch size is increased. In contrast, the computation time grows more with the filter size, as a larger filter increases the size of matrix equations in Eq. (3) quadratically over $n = c_i \times h_f \times w_f$. It implies that the proposed method enables training with larger batch sizes within fixed memory constraints, allowing memory-limited systems to utilize increased batch sizes and thus reduce the required training epochs. For example, training ResNeXt-50 (Xie et al., 2017) on ImageNet dataset with a batch size of 64, standard convolutions require 3.4 GB activation memory. In contrast, the proposed method reduces it to 1.6 GB, enabling nearly double the batch size given the same memory budget.

6 DISCUSSION

Compatibility and Scalability. The proposed method is widely compatible with diverse CNN variants by leveraging their common stride-wise kernel application, which significantly improves memory efficiency across these structures. Furthermore, the core principle of parameter sharing is scalable to Transformers, suggesting it could also reduce activation memory in those models, although optimizing matrix size and multiplication is necessary for practical implementation.

Implementation and Comparison. The current custom CUDA implementation of ECAI introduces a time overhead compared to standard optimized PyTorch. However, ECAI provides a memory-accuracy trade-off, achieving lower memory consumption than Gradient Checkpointing and maintaining model accuracy (unlike Activation Quantization), making it ideal for memory-constrained environments.

For a detailed analysis of all implementation over-

heads, specific comparative results and further technical viewpoints, please refer to the Appendix A.

7 CONCLUSION

We introduce a constant activation memory CNN training, which leverages efficient convolution activation inversion to reconstruct activations without storing them during backpropagation. It significantly reduces memory usage while maintaining equivalent model performance, addressing the critical challenge of memory demands during CNN training. The proposed mechanism efficiently solves n matrix equations in parallel with $\mathcal{O}(n^2)$, a notable improvement over standard methods like Gaussian-Jordan elimination, which operates at $\mathcal{O}(n^4)$. Implemented using CUDA on memory-constrained platforms such as Jetson Nano, it demonstrates the practicality through extensive experiments across various models and tasks, achieving constant activation memory usages. Overall, the proposed method facilitates the effective training of CNNs for advanced vision tasks by enabling constant memory usage, allowing for scaling image sizes and CNN architectures in real-world applications. It can also be extended to other model like ViT to reduce activation memory caused by repeatedly applied parameters.

References

O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

H. Anton and C. Rorres. *Elementary linear algebra: applications version*. John Wiley & Sons, 2013.

K. Atkinson. *An introduction to numerical analysis*. John wiley & sons, 1991.

Y. Bengio, I. Goodfellow, and A. Courville. *Deep learning*, volume 1. MIT press Cambridge, MA, USA, 2017.

D. Bershatsky, A. Mikhalev, A. Katrutsa, J. Gusak, D. Merkulov, and I. Oseledets. Memory-efficient backpropagation through large linear layers, 2022. URL `https://arxiv.org/abs/2201.13195`.

A. Chakrabarti and B. Moseley. Backprop with approximate activations for memory-efficient network training, 2019. URL `https://arxiv.org/abs/1901.07988`.

L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation, 2017. URL `https://arxiv.org/abs/1706.05587`.

T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020.

N. Corporation. *NVIDIA RTX A6000 GPU*, 2020. URL `https://www.nvidia.com/en-us/design-visualization/rtx-a6000/`. Accessed: 2024-02-13.

I. Demir, K. Koperski, D. Lindenbaum, G. Pang, J. Huang, S. Basu, F. Hughes, D. Tuia, and R. Raskar. Deepglobe 2018: A challenge to parse the earth through satellite images. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, page 172–17209. IEEE, June 2018. doi: 10.1109/cvprw.2018.00031. URL `http://dx.doi.org/10.1109/CVPRW.2018.00031`.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

A. Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

R. D. Evans and T. Aamodt. Ac-gc: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems*, 34:27434–27448, 2021.

M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

GigaPan. GigaPan High-Resolution Imagery. URL `https://gigapan.com`. Accessed: Date.

I. Gohberg, T. Kailath, and V. Olshevsky. Fast gaussian elimination with partial pivoting for matrices with displacement structure. *Mathematics of computation*, 64(212):1557–1576, 1995.

A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. The reversible residual network: Backpropagation without storing activations. *Advances in neural information processing systems*, 30, 2017.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. *Advances in neural information processing systems*, 29, 2016.

K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, et al. A survey on vision transformer. *IEEE transactions on*

pattern analysis and machine intelligence, 45(1):87–110, 2022.

T. Hascoet, Q. Febvre, W. Zhuang, Y. Ariki, and T. Takiguchi. Reversible designs for extreme memory cost reduction of cnn training. *EURASIP Journal on Image and Video Processing*, 2023(1):1, 2023.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn, 2018. URL `https://arxiv.org/abs/1703.06870`.

J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

J. Ho, T. Salimans, A. Gritsenko, W. Chan, M. Norouzi, and D. J. Fleet. Video diffusion models. *Advances in Neural Information Processing Systems*, 35:8633–8646, 2022.

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

J.-H. Jacobsen, A. Smeulders, and E. Oyallon. i-revnet: Deep invertible networks. *arXiv preprint arXiv:1802.07088*, 2018.

H. J. Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

R. Khanam and M. Hussain. What is yolov5: A deep look into the internal features of the popular object detector, 2024. URL `https://arxiv.org/abs/2407.20892`.

Ç. K. Koç and S. N. Arachchige. A fast algorithm for gaussian elimination over gf (2) and its implementation on the gapp. *Journal of Parallel and Distributed Computing*, 13(1):118–122, 1991.

A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

R. Kumar, M. Purohit, Z. Svitkina, E. Vee, and J. Wang. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

C. Lee and S. Lee. Softmax output approximation for activation memory-efficient training of attention-based networks. *Advances in Neural Information Processing Systems*, 36, 2024.

S. J. Leon, L. De Pillis, and L. G. De Pillis. *Linear algebra with applications*. Pearson Prentice Hall Upper Saddle River, NJ, 2006.

Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 33(12):6999–7019, 2021.

T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context, 2015. URL `https://arxiv.org/abs/1405.0312`.

T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection, 2017. URL `https://arxiv.org/abs/1612.03144`.

B. Liu, X. Yu, A. Yu, P. Zhang, G. Wan, and R. Wang. Deep few-shot learning for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 57(4):2290–2304, 2018a.

S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia. Path aggregation network for instance segmentation, 2018b. URL `https://arxiv.org/abs/1803.01534`.

X. Liu, L. Zheng, D. Wang, Y. Cen, W. Chen, X. Han, J. Chen, Z. Liu, J. Tang, J. Gonzalez, M. Mahoney, and A. Cheung. Gact: Activation compressed training for generic network architectures, 2022a. URL `https://arxiv.org/abs/2206.11357`.

Z. Liu, K. Zhou, F. Yang, L. Li, R. Chen, and X. Hu. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2021.

Z. Liu, K. Zhou, F. Yang, L. Li, R. Chen, and X. Hu. EXACT: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, 2022b. URL `https://openreview.net/forum?id=vkaMaq95_rX`.

J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation, 2015. URL `https://arxiv.org/abs/1411.4038`.

D. Luo and X. Wang. Moderntcn: A modern pure convolution structure for general time series analysis. In *The Twelfth International Conference on Learning Representations*, 2024.

E. Maggiori, Y. Tarabalka, G. Charpiat, and P. Alliez. Can semantic labeling methods generalize to

any city? the inria aerial image labeling benchmark. In *2017 IEEE International geoscience and remote sensing symposium (IGARSS)*, pages 3226–3229. IEEE, 2017.

J. Martens and I. Sutskever. Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade: Second Edition*, pages 479–535. Springer, 2012.

NASA. NASA Earthdata. URL `https://earthdata.nasa.gov`. Accessed: Date.

NVIDIA. Jetson Nano Developer Kit, 2019. URL `https://developer.nvidia.com/\\embedded/jetson-nano-developer-kit`. Accessed: Date.

NVIDIA Corporation. NVIDIA GeForce RTX 3090. `https://www.nvidia.com/en-us/geforce/\\graphics-cards/30-series/rtx-3090`, 2020.

T. O'Haver. Intro to signal processing-deconvolution. *University of Maryland at College Park, Retrieved*, 2008.

J. Panda and S. Meher. Recent advances in 2d image upscaling: A comprehensive review. *SN Computer Science*, 5(6):735, 2024.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár. Designing network design spaces, 2020. URL `https://arxiv.org/abs/2003.13678`.

M. Rahnemoonfar, T. Chowdhury, A. Sarkar, D. Varshney, M. Yari, and R. R. Murphy. Floodnet: A high resolution aerial imagery dataset for post flood scene understanding. *IEEE Access*, 9: 89644–89654, 2021.

J. Redmon. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: towards real-time object detection with region proposal networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 91–99, Cambridge, MA, USA, 2015. MIT Press.

R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015a.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015b. URL `https://arxiv.org/abs/1505.04597`.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. URL `https://arxiv.org/abs/1801.04381`.

V. G. Satorras and J. B. Estrach. Few-shot learning with graph neural networks. In *International conference on learning representations*, 2018.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. URL `https://arxiv.org/abs/1409.1556`.

V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5):1299–1312, 2016.

H. Wang, S. Ge, Z. Lipton, and E. P. Xing. Learning robust global representations by penalizing local predictive power. In *Advances in Neural Information Processing Systems*, pages 10506–10518, 2019.

N. Wiener. *Extrapolation, interpolation, and smoothing of stationary time series: with engineering applications*. The MIT press, 1949.

S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks, 2017. URL `https://arxiv.org/abs/1611.05431`.

M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on computer vision and pattern recognition*, pages 2528–2535. IEEE, 2010.

B. Zhao, H. Lu, S. Chen, J. Liu, and D. Wu. Convolutional neural networks for time series classification. *Journal of systems engineering and electronics*, 28 (1):162–169, 2017a.

H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network, 2017b. URL `https://arxiv.org/abs/1612.01105`.

Z. Zhou, J. Shin, L. Zhang, S. Gurudu, M. Gotway, and J. Liang. Fine-tuning convolutional neural networks for biomedical image analysis: actively and incrementally. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7340–7351, 2017.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [Yes]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes]
2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [Not Applicable]
   (b) Complete proofs of all theoretical results. [Not Applicable]
   (c) Clear explanations of any assumptions. [Not Applicable]
3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [Yes]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes]
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [Yes]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. [Not Applicable]
   (b) The license information of the assets, if applicable. [Not Applicable]
   (c) New assets either in the supplemental material or as a URL, if applicable. [Not Applicable]
   (d) Information about consent from data providers/curators. [Not Applicable]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [Not Applicable]
5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [Not Applicable]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [Not Applicable]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [Not Applicable]

ECAI: Efficient Convolution Activation Inversion for Constant-Memory Convolutional Neural Networks Training: Supplementary Materials

A Discussions and Limitations

Compatibility. The proposed method significantly improves memory efficiency and can be applied to various types of convolution layers. Recent architectures employ diverse variants such as depthwise separable, transposed, dilated, and dynamic convolutions. As long as these variants retain the core characteristic of standard convolution—where a kernel is repeatedly applied in a stride-wise manner over a single input—the proposed method can be effectively used to reduce memory consumption. In particular, for architectures with deeply stacked convolution layers, it can further alleviate memory bottlenecks. For instance, in depthwise separable convolutions, the method can be applied to the point-wise convolution to reduce memory usage. In dilated convolutions, additional savings can be achieved by reconstructing only the input elements actually used in computation. Thus, the proposed method contributes to improved activation memory efficiency across a wide range of convolutional structures.

Scalability. Although, the proposed method primarily targets convolution layers, a similar pattern of repeated parameter usage also exists in Transformers. In multi-head self-attention, the same key, query, and value weight matrices are applied token-wise across the input sequence, with the number of tokens increasing proportionally to the input length (e.g., the number of patches in ViT). While the resulting inverse coefficient matrix may become large, this shared structure suggests that our method could be extended to reduce intermediate activation memory in Transformer models. Furthermore, optimizing or simplifying the matrix multiplications between token representations and weight matrices may help keep the inverse coefficient matrix size within a practical range, enabling the scalable use of our method.

Type	Gradient computation		Optimizer	Our method	
Kernel	cudnn::wgrad	cudnn::dgrad	cutlass::grad	Inversion	Additional process
Time (ms)	0.172	0.267	1.147	60.566	66.781

Table 7: Average execution time of CUDA kernels during ResNet-50 training, analyzed using NVIDIA Nsight Systems.

Implementation. While the proposed method effectively reduces memory usage, its current implementation is not yet as optimized as standard PyTorch operations that leverage highly-tuned CUDA libraries such as CUTLASS or cuDNN. As shown in Table 7, the inversion process introduces notable time overhead compared to operations with theoretical complexity around $\mathcal{O}(n^{2\sim3})$, including gradient computation and optimizer updates. Additional delays arise from input reordering and I/O between PyTorch extensions and custom CUDA kernels, which are comparable in cost to the inversion itself. Despite these limitations, we believe the memory–computation trade-off is acceptable in practice, especially in memory-constrained environments such as edge devices or high-resolution tasks like satellite or pathology imaging. In such cases, enabling training with moderate time overhead is often preferable to being unable to train the model at all. We are actively optimizing the implementation to reduce computational overhead and enhance the practical usability of the proposed method.

Gradient reconstruction error. As shown in Fig. 9, we used the proposed method ECAI to evaluate the impact of reconstructed input activations—obtained at each layer through a 10-layer convolutional network with

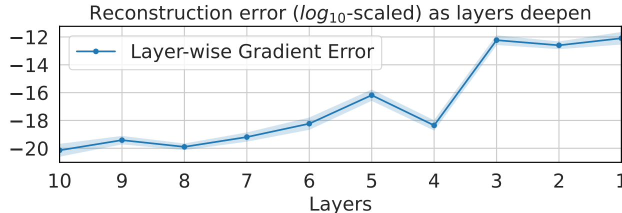

Figure 9: Layer-wise comparison of accumulated gradient reconstruction error (MSE) between our method and the original convolutional layers without reconstruction, in a 10-layer CNNs. The graph shows the mean and standard deviation across 10 runs, with shaded regions indicating upper and lower bounds.

inverse coefficient matrix sizes up to $n = 1000$—on the gradient matrix. The reconstruction error after all 10 layers, measured by mean squared error (MSE), was found to be near zero (approximately 10^{-12}), which is comparable to the precision limits of standard quantization techniques and 32-bit floating-point arithmetic. Although such accumulated gradient errors may vary in deeper convolutional networks, our experiments demonstrate that these small errors do not degrade training accuracy or hinder convergence when using ECAI.

Method	Baseline	Gradient Checkpointing	Activation Quantization	ECAI
Training time (sec)	174.37	175.10	174.38	217.23
Memory reduction (MB)	-	272.75	158.61	281.3
Accurcy (%)	84.05	84.33	73.19	84.49

Table 8: Comparative Evaluation of ECAI against Gradient Checkpointing and Activation Quantization. Performance metrics, including training time, memory reduction, and accuracy, are evaluated by training bottleneck-structured CNN models on the CIFAR-10 dataset for 30 epochs.

Comparative Evaluation of Memory Efficiency and Accuracy. We evaluated the proposed method against common memory optimization techniques, namely Gradient Checkpointing (GC) and Activation Quantization (AQ), by training common bottleneck-structured CNN models on the CIFAR-10 dataset for 30 epochs (Table 8). Despite the minimal time overhead incurred by our current custom implementation, the results show that our proposed method offers memory efficiency compared to GC, which introduces a substantial memory overhead due to storing the computation graph for recomputation. In contrast, ECAI, with its constant memory approach, achieves a lower effective activation memory footprint. Crucially, while our method exhibited a 1.5× increase in training time compared to GC (a limitation we attribute primarily to current implementation overheads rather than fundamental algorithmic complexity), it provides a significant advantage in accuracy over Activation Quantization. Unlike AQ, which inevitably sacrifices model precision by reducing activation bit-width, the high fidelity of ECAI's activation reconstruction ensures that model accuracy is maintained or even slightly improved, demonstrating a more favorable trade-off between memory and performance.

B Additional Related works

Reversible Networks. Some networks, such as RevNet (Gomez et al., 2017), iRevNet (Jacobsen et al., 2018), and iResNet (Hascoet et al., 2023), reduce activation memory usage by leveraging structural invertibility. Most reversible networks are based on the residual bottleneck structure of ResNet (He et al., 2015), employing a transformation that allows the inversion of sequential layer operations in a linear-like manner to reduce activation memory usage. However, RevNet (Gomez et al., 2017) does not achieve full invertibility due to non-linear operations such as batch norm, while iRevNet (Jacobsen et al., 2018) ensures complete invertibility by eliminating batch norm, stride, and pooling, at the cost of reduced model expressiveness. iResNet (Hascoet et al., 2023), in contrast, maintains invertibility only under specific conditions, such as Lipschitz constraints. These approaches inherently introduce unquantifiable computational overhead for inversion and remain highly architecture-dependent. In contrast, the proposed activation inverse offers equivalent model performance through its efficient and exact inversion, applicable to a wide range of CNNs.

Low Activation Memory Training. Several techniques have been proposed to reduce activation memory, such as re-materialization (Kumar et al., 2019; Chen et al., 2016; Gruslys et al., 2016), checkpointing (Martens and Sutskever, 2012), activation compression (Evans and Aamodt, 2021; Liu et al., 2021, 2022a), and approximation (Lee and Lee, 2024; Chakrabarti and Moseley, 2019). Re-materialization reduces activation memory consumption by recomputing activations during back-propagation instead of storing them. However, this approach requires multiple re-executions of the network, resulting in a high computational overhead of $\mathcal{O}(L^2n^3b)$ for an L-layer network with feature size n and batch size b. In contrast, the proposed method reduces activation memory with a significantly lower overhead of $\mathcal{O}(Ln^2b)$. Alternatively, activation compression and approximation reduce memory usage by leveraging methods such as sparsity, compression coding, quantization, and low-rank approximation. However, these approaches result in the loss of activation values and gradients, leading to potential degradation in model performance. Consequently, these methods have a fundamental limitation where model convergence must be sacrificed to achieve memory reduction. In contrast, the proposed method reconstructs activations exactly, ensuring memory efficiency without compromising model performance.

Deconvolution. Deconvolution has been used in a different context from its mathematical definition in signal processing (O'Haver, 2008; Wiener, 1949), where deconvolution in signal processing refers to the process of inverting features in the Laplace (frequency) domain back into time-domain signals. Due to the computational complexity of conventional deconvolution when applied to high-dimensional data, deconvolutional networks (Zeiler et al., 2010) in deep learning leverage the concept of deconvolution by utilizing a sparsity prior to optimize feature maps and filters, enabling diverse reconstructions. Consequently, deconvolution in deep learning is primarily employed for visualization or upsampling (Panda and Meher, 2024) rather than for precise mathematical inversion. However, this concept of deep learning deconvolution does not conflict with the proposed convolution activation inverse, which analytically derives the exact inverse and applies it to the deep learning training process. Notably, the proposed method is the first that accelerates the inversion process for convolution (deconvolution), realizing its conceptual application.

C Gauss-Jordan Elimination (GJE)

Gauss-Jordan elimination (Atkinson, 1991) is a method that solves a system of linear equations (Anton and Rorres, 2013) by transforming a matrix equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{b}, \mathbf{x} \in \mathbb{R}^{n\times 1}$, into the reduced row echelon form (Leon et al., 2006). Given n equations for n unknowns, it solves $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ with the computation complexity of $\mathcal{O}(n^3)$, as illustrated in Fig. 10.

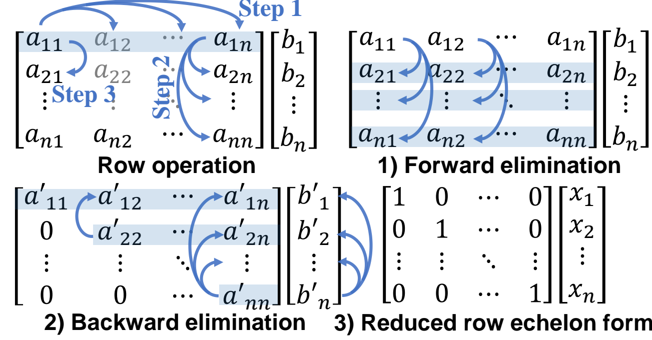

Figure 10: The procedure of Gauss-Jordan elimination (Atkinson, 1991) that solves $\mathbf{Ax} = \mathbf{b}$ in $\mathcal{O}(n^3)$ via forward and back elimination with row-wise operations to derive the reduced row echelon form (Leon et al., 2006).

Row-wise Operation. Gauss-Jordan elimination applies three steps of row-wise operations in its forward and backward elimination: [step 1] $\mathbf{A}_i \rightarrow k\mathbf{A}_i$ when $k \neq 0$, [step 2] $\mathbf{A}_i \rightarrow \mathbf{A}_i + k\mathbf{A}_j$ when $i \neq j$, and [step 3] $\mathbf{A}_i \leftrightarrow \mathbf{A}_j$ otherwise, where $\mathbf{A}_i$ and $\mathbf{A}_j$ is the i-th and j-th row of the coefficient matrix $\mathbf{A}$, respectively, and k is a constant.

1) Forward Elimination (Fig. 10.1): Forward elimination performs three row-wise operations (i.e., step 1, 2, and 3) to transform the coefficient matrix $\mathbf{A}$ into an upper triangular form by setting the pivot of each selected column to one and making all entries below it zero. This procedure is repeated until the matrix attains an upper triangular form by adjusting $n - i$ elements in the i-th row. As row-wise operations are applied across all elements for each column, a total of $n + 1$ operations is performed for each row at each step, which requires $\sum_{i=1}^{n}(n - i)(n + 1) = \mathcal{O}(n^3)$ operations in total.

2) Backward Elimination (Fig. 10.2): Backward elimination performs three row-wise operations (i.e., step 1, 2, and 3) to transform the upper triangular matrix obtained through forward elimination into an identity matrix by sequentially progressing from the last row to the first. This process requires a total of $\sum_{i=1}^{n}(n - i) = \mathcal{O}(n^2)$ operations.

3) Reduced Row Echelon Form (Fig. 10.3): Finally, the solution to the matrix equation $\mathbf{Ax} = \mathbf{b}$ is obtained as $\mathbf{x} = [x_1, \cdots, x_n]$ from the reduced row echelon form of the matrix leon2006linear derived from the backward elimination.

Computation Complexity. Consequently, the overall computation complexity of Gauss-Jordan elimination approaches $\mathcal{O}(n^3)$ as shown below:

$$\overbrace{\sum_{i=1}^{n}(n-i)(n+1)}^{\text{Forward elimination}=\mathcal{O}(n^3)} + \overbrace{\sum_{i=1}^{n}(n-i)}^{\text{Backward elimination}=\mathcal{O}(n^2)} = \mathcal{O}(n^3) \tag{6}$$

While some studies propose slight reductions in the complexity of solving a single matrix equation, these reductions are insignificant, such as from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{2.81})$ using the Strassen algorithm (Strassen, 1969), and are often restricted to specific matrix types, like Toeplitz matrices (Gohberg et al., 1995; Koç and Arachchige, 1991). Consequently, the cubic complexity growth when solving a matrix equation is challenging even for small values of n.

D Experiment setups

Experiments were conducted on both A6000 (48GB) and RTX 3090 (24GB) GPUs using a CUDA 11.8, PyTorch 1.14.0, and TorchVision 0.15.0 environment. Custom CUDA kernels were implemented and integrated via PyTorch extensions. For model-level evaluation, pretrained weights from TorchVision and Ultralytics (v8.3.28) were used. Datasets were loaded using TorchVision, and the ImageNet-Sketch dataset was obtained from a public GitHub repository (Wang et al., 2019). On the edge device Jetson Nano (4GB), experiments were conducted using the same methodology under JetPack 4.6.1 with PyTorch 1.10.0a0. To measure training time and memory usage, custom convolution layers were initialized with random parameters. For parallelization in the proposed ECAI method, appropriate thread counts (1024 and 256) were selected depending on the device.

The code below reconstructs the activation memory of the proposed ECAI method and passes it to PyTorch in the form of a PyTorch tensor. Further details and examples can be found via the GitHub[2].

```
#include <torch/extension.h>
#include <vector>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cooperative_groups.h>

namespace cg = cooperative_groups;

namespace {
    # The 4D convolution filter is reshaped into a 2D matrix.
    __global__ void flatten_filter(const double *original_filter, double *
        flatten_filter, double *seperate_filter,
        int output_channel, int kernel_height, int kernel_width,
        int gap, int number_of_each_conv, int flag) {

        int channel_idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (channel_idx < number_of_each_conv * output_channel) {
            int idx_x = channel_idx / output_channel;
            int idx_y = channel_idx % output_channel;

            int w = idx_x / (kernel_height * kernel_width);
            int remainder = idx_x % (kernel_height * kernel_width);
            int h = remainder / kernel_height;
            int o = remainder % kernel_height;
            int o_idx = (idx_y * number_of_each_conv) + (w * kernel_height *
                kernel_width) + (h * kernel_width) + o;

            if (flag == 1) {
                if (idx_x < output_channel && idx_y < output_channel) {
                    int f_idx = (idx_y * output_channel) + idx_x;
                    flatten_filter[f_idx] = original_filter[o_idx];
                } else {
                    int f_idx = (idx_y * gap) + (idx_x - output_channel);
                    seperate_filter[f_idx] = original_filter[o_idx];
                }
                __syncthreads();
            } else {
                if(idx_x < number_of_each_conv && idx_y < number_of_each_conv) {
                    int f_idx = (idx_y * number_of_each_conv) + idx_x;
                    flatten_filter[f_idx] = original_filter[o_idx];
                }
                __syncthreads();
            }
            __syncthreads();
```

[2] https://github.com/eai-lab/ECAI

```
    }
}
# We selectively identify the regions that cannot be inverted through our method.
__global__ void select_input(const float *input, float *saved_input,
    int output_height, int output_width,
    int input_channel, int input_height, int input_width,
    int kernel_height, int kernel_width, int stride,
    int w_number_of_patch_input,
    int high, int gap, int number_of_each_conv,
    int total_patch, int batch_size) {

    int channel_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int batch_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (channel_idx < high && batch_idx < batch_size) {
        for (int patch_idx = 0; patch_idx < total_patch; patch_idx++) {
            int patch_height = (patch_idx / w_number_of_patch_input) * (
                kernel_height / stride);
            int patch_width = (patch_idx % w_number_of_patch_input) * (
                kernel_width / stride);

            if (patch_height > output_height-1) patch_height = output_height-1;
            if (patch_width > output_width-1) patch_width = output_width-1;

            int in_ch = channel_idx/(kernel_height*kernel_width);
            int in_h = ((channel_idx % (kernel_height*kernel_width))/kernel_width)
                 + patch_height*stride;
            int in_w = ((channel_idx % (kernel_height*kernel_width))%kernel_width)
                 + patch_width*stride;
            int idx = (batch_idx * input_channel * input_height * input_width) + (
                in_ch * input_height * input_width) + (in_h* input_width) + in_w;

            if (channel_idx > number_of_each_conv-gap-1) {
                saved_input[(batch_idx*total_patch*gap)+(patch_idx*gap) + (
                    channel_idx-(number_of_each_conv-gap))] = input[idx];
            }
            __syncthreads();
        }
    }
}

# To ensure the accuracy of the reconstructed activations, the values
    corresponding to the selected inputs are removed.
__global__ void constant_ell(
    const float *saved_input, double *output, const double *
        seperate_flatten_filter,
    int output_channel, int output_height, int output_width,
    int w_number_of_patch_input, int kernel_height, int kernel_width, int stride,
    int gap, int total_patch, int batch_size) {

    int x_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int batch_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (x_idx < output_channel && batch_idx < batch_size) {
        for (int patch_idx = 0; patch_idx < total_patch; patch_idx++) {
            float sum = 0.0f;
            for (int k = 0 ; k < gap; k++){
                sum += seperate_flatten_filter[x_idx*gap+k]*saved_input[(batch_idx
                    *gap*total_patch)+(patch_idx*gap)+k];
            }
            __syncthreads();

            int patch_height = (patch_idx / w_number_of_patch_input) * (
                kernel_height / stride);
            int patch_width = (patch_idx % w_number_of_patch_input) * (
                kernel_width / stride);
```

```
                if (patch_height > output_height-1) patch_height = output_height-1;
                if (patch_width > output_width-1) patch_width = output_width-1;

                int output_idx = batch_idx*(output_channel*output_height*output_width)
                    + x_idx*(output_height*output_width)+(patch_height*output_width)+
                    patch_width;
                output[output_idx] -= sum;
                __syncthreads();
            }
            __syncthreads();
        }
    }
    # The value altered by the bias is removed.
    __global__ void bias_ell(
        const float *bias, double *output,
        int output_channel, int output_height, int output_width,
        int w_number_of_patch_input, int kernel_height, int kernel_width, int stride,
        int total_patch, int batch_size) {

        int x_idx = blockIdx.x * blockDim.x + threadIdx.x;
        int batch_idx = blockIdx.y * blockDim.y + threadIdx.y;

        if (x_idx < output_channel && batch_idx < batch_size) {
            for (int patch_idx = 0; patch_idx < total_patch; patch_idx++) {

                int patch_height = (patch_idx / w_number_of_patch_input) * (
                    kernel_height / stride);
                int patch_width = (patch_idx % w_number_of_patch_input) * (
                    kernel_width / stride);
                if (patch_height > output_height-1) patch_height = output_height-1;
                if (patch_width > output_width-1) patch_width = output_width-1;
                int output_idx = batch_idx*(output_channel*output_height*output_width)
                    + x_idx*(output_height*output_width)+(patch_height*output_width)+
                    patch_width;

                output[output_idx] -= bias[x_idx];
                __syncthreads();
            }
            __syncthreads();
        }
    }
    # A parallelized inversion method is performed for ECAI.
    __global__ void GE_under(double *filter, double *matrix, int low){
        const uint x_idx = blockIdx.x * blockDim.x + threadIdx.x;
        int row, column;
        if (x_idx < low) {
            matrix[x_idx * low + x_idx ] = 1.0;
            __syncthreads();
            for (row = 0; row < low; row++) {
                const double diag = filter[row*low+row];
                if (diag == 0.0f) continue;
                __syncthreads();
                matrix[row*low + x_idx] /= diag;
                filter[row*low + x_idx] /= diag;
                __threadfence();

                for (column = 0; column < low; column++) {
                    if (column == row ) continue;
                    const double pivot = filter[column * low + row];
                    __syncthreads();
                    matrix[column * low + x_idx] -= matrix[row * low + x_idx] * pivot;
                    filter[column * low + x_idx] -= filter[row * low + x_idx] * pivot;
                    __threadfence();
                }
            }
```

```
    }
}
# If the size exceeds the maximum allowed for parallelization, the process is
    carried out sequentially using a more stable method.
__global__ void GE_over(double *filter, double *matrix, int low){
    cg::grid_group grid = cg::this_grid();
    const uint x_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int row, column;
    if (x_idx < low) {
        matrix[x_idx * low + x_idx ] = 1.0;
        grid.sync();
        __syncthreads();
        for (row = 0; row < low; row++) {
            const double diag = filter[row*low+row];
            if (diag == 0.0f) continue;
            grid.sync();
            __syncthreads();
            matrix[row*low + x_idx] /= diag;
            filter[row*low + x_idx] /= diag;
            __threadfence();

            for (column = 0; column < low; column++) {
                if (column == row ) continue;
                const double pivot = filter[column * low + row];
                grid.sync();
                __syncthreads();
                matrix[column * low + x_idx] -= matrix[row * low + x_idx] * pivot;
                filter[column * low + x_idx] -= filter[row * low + x_idx] * pivot;
                __threadfence();
            }
        }
    }
}
# Using the computed coefficient inverse matrix, all activation values are
    reconstructed through simple operations while preserving the original spatial
    information.
__global__ void align_output(double* output, double* matrix, float* d_saved_input,
     float* d_inversion_list,
    int output_channel, int output_height, int output_width, int kernel_height,
        int kernel_width,
    int stride, int w_number_of_patch_input,
    int input_channel, int input_height, int input_width,
    int total_patch, int low, int high, int gap, int batch_size){

    int channel_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int batch_idx = blockIdx.y * blockDim.y + threadIdx.y;

    if (channel_idx < high && batch_idx < batch_size) {
        for (int patch_idx = 0; patch_idx < total_patch; patch_idx++) {
            int patch_height = (patch_idx / w_number_of_patch_input) * (
                kernel_height / stride);
            int patch_width = (patch_idx % w_number_of_patch_input) * (
                kernel_width / stride);

            if (patch_height > output_height-1) patch_height = output_height-1;
            if (patch_width > output_width-1) patch_width = output_width-1;

            int in_ch = channel_idx/(kernel_height*kernel_width);
            int in_h = ((channel_idx % (kernel_height*kernel_width))/kernel_width)
                 + patch_height*stride;
            int in_w = ((channel_idx % (kernel_height*kernel_width))%kernel_width)
                 + patch_width*stride;

            int idx = (batch_idx * input_channel * input_height * input_width) + (
                in_ch * input_height * input_width) + (in_h* input_width) + in_w;
```

```
                if (channel_idx < low) {
                    double sum = 0.0;
                    for (int i = 0; i < low; i ++) {
                        int output_idx = (batch_idx*output_channel*output_height*
                            output_width)+ (i*output_height*output_width)+(
                            patch_height*output_width)+patch_width;
                        sum += matrix[channel_idx*low + i] * output[output_idx];
                    }
                    d_inversion_list[idx] = static_cast<float>(sum);
                } else {
                    if (gap > 0) {
                        d_inversion_list[idx] = static_cast<float>(d_saved_input[(
                            batch_idx*gap*total_patch) + (gap*patch_idx) + (
                            channel_idx-low)]);
                    }
                }
                __syncthreads();
            }
            __syncthreads();
        }
    }
}

torch::Tensor inversion_forward(
    const torch::Tensor input,
    const torch::Tensor values) {

    auto input_double = input.to(torch::kFloat);
    float* d_input = input_double.data_ptr<float>();

    int batch_size       = input.size(0);
    int input_channel    = input.size(1);
    int input_height     = input.size(-2);
    int input_width      = input.size(-1);

    int stride           = values[0].item<int>();
    int output_channel   = values[1].item<int>();
    int output_height    = values[2].item<int>();
    int output_width     = values[3].item<int>();
    int kernel_height    = values[4].item<int>();
    int kernel_width     = values[5].item<int>();
    int padding          = values[6].item<int>(); //

    int h_number_of_patch_input = (input_height-padding)  / ((kernel_height / stride)
       * stride);
    int w_number_of_patch_input = (input_width-padding)   / ((kernel_width / stride) *
        stride);
    # To efficiently compute the inversion patches, a patch selection process is
       performed.
    if (input_height-(padding)    > ((h_number_of_patch_input-1) *((kernel_height /
       stride) * stride))+kernel_height) h_number_of_patch_input =
       h_number_of_patch_input+1;
    if (input_width-(padding)     > ((w_number_of_patch_input-1) *((kernel_width /
       stride)  * stride))+kernel_width) w_number_of_patch_input =
       w_number_of_patch_input+1;

    int total_patch = h_number_of_patch_input * w_number_of_patch_input;

    const int number_of_each_conv = input_channel*kernel_height*kernel_width;

    int gap = number_of_each_conv-output_channel;
    if (gap < 0) gap = 0;

    int high = output_channel;
    if (output_channel < number_of_each_conv) high = number_of_each_conv;
```

```
    float* d_saved_input = nullptr;
    size_t size_saved_input = batch_size * gap * total_patch * sizeof(float);
    cudaMalloc(&d_saved_input, size_saved_input);

    dim3 threadsPerBlock(1024, 1);
    const int num_block_high = (high + threadsPerBlock.x - 1) / threadsPerBlock.x;
    dim3 numblocks_high(num_block_high, batch_size);

    if (gap > 0) {
        select_input<<<numblocks_high, threadsPerBlock>>>(
            d_input, d_saved_input,
            output_height, output_width,
            input_channel, input_height, input_width,
            kernel_height, kernel_width, stride,
            w_number_of_patch_input,
            high, gap, number_of_each_conv,
            total_patch, batch_size
        );
        cudaDeviceSynchronize();
    }

    torch::Tensor data_tensor = torch::from_blob(
        d_saved_input,
        {batch_size, total_patch, gap},
        torch::TensorOptions().dtype(torch::kFloat32).device(torch::kCUDA)
    ).clone();

    cudaFree(d_saved_input);
    return data_tensor;
}

torch::Tensor inversion_backward(
  const torch::Tensor filter,
  const torch::Tensor output,
  const torch::Tensor input,
  const torch::Tensor values,
  const torch::Tensor bias) {

    auto weight_double = filter.to(torch::kDouble);
    auto output_double = output.to(torch::kDouble);
    auto input_float = input.to(torch::kFloat);
    auto bias_float = bias.to(torch::kFloat);

    double* d_filter = weight_double.data_ptr<double>();
    double* d_output = output_double.data_ptr<double>();
    float* d_input = input_float.data_ptr<float>();
    float* d_bias = bias_float.data_ptr<float>();

    int stride = values[0].item<int>();
    int padding = values[1].item<int>();
    int dilation = values[2].item<int>();
    int bias_bool = values[3].item<int>();
    int input_width = values[4].item<int>();
    int input_height = values[5].item<int>();

    int output_channel   = filter.size(0);
    int output_height    = output.size(-2);
    int output_width     = output.size(-1);

    int kernel_height    = filter.size(-2);
    int kernel_width     = filter.size(-1);
    int batch_size       = output.size(0);

    int input_channel    = filter.size(1);

    int h_number_of_patch_input = (input_height-padding)  / ((kernel_height / stride)
```

```
    * stride);
int w_number_of_patch_input = (input_width-padding)   / ((kernel_width / stride) *
     stride);

if (input_height-padding    > ((h_number_of_patch_input-1) *((kernel_height /
    stride) * stride))+kernel_height) h_number_of_patch_input =
    h_number_of_patch_input+1;
if (input_width-padding     > ((w_number_of_patch_input-1) *((kernel_width /
    stride)  * stride))+kernel_width) w_number_of_patch_input =
    w_number_of_patch_input+1;

int total_patch = h_number_of_patch_input * w_number_of_patch_input;

const int number_of_each_conv = input_channel*kernel_height*kernel_width;

int gap = number_of_each_conv-output_channel;
if (gap < 0) gap = 0;

int high = output_channel;
if (output_channel < number_of_each_conv) high = number_of_each_conv;

int low = output_channel;
if (output_channel > number_of_each_conv) low = number_of_each_conv;

int conv_high_flag;
if (output_channel < number_of_each_conv) conv_high_flag = 1;
else conv_high_flag = 0;
int patch_idx_num = (total_patch / low) + 1;

double* d_ge_filter;
size_t size_flatten_ge_filter = low * low * sizeof(double);
cudaMalloc((void**)&d_ge_filter, size_flatten_ge_filter);

double* d_matrix;
cudaMalloc((void**)&d_matrix, size_flatten_ge_filter);
cudaMemset(d_matrix, 0.0, size_flatten_ge_filter);

double* d_saved_filter = nullptr;
size_t size_flatten_saved_filter = output_channel * gap * sizeof(double);
cudaMalloc((void**)&d_saved_filter, size_flatten_saved_filter);

float* d_inversion_input;
size_t size_inversion_input = input_channel * batch_size * input_height *
    input_width * sizeof(float);
cudaMalloc((void**)&d_inversion_input, size_inversion_input);

dim3 threadsPerBlock(1024);

const int num_block_high = (high + threadsPerBlock.x - 1) / threadsPerBlock.x;
const int num_block_filter = ((output_channel*number_of_each_conv) +
    threadsPerBlock.x - 1) / threadsPerBlock.x;
const int num_block = ((low) + threadsPerBlock.x - 1) / threadsPerBlock.x;

dim3 numblocks(num_block, 1);
dim3 numblocks_high(num_block_high, batch_size);
dim3 numblocks_filter(num_block_filter, 1);

flatten_filter<<<numblocks_filter, threadsPerBlock>>>(
    d_filter, d_ge_filter, d_saved_filter,
    output_channel, kernel_height, kernel_width,
    gap, number_of_each_conv, conv_high_flag
    );
cudaDeviceSynchronize();

if (bias_bool == 1) {
    bias_ell<<<numblocks_high, threadsPerBlock>>>(
```

```
            d_bias, d_output,
            output_channel, output_height, output_width,
            w_number_of_patch_input, kernel_height, kernel_width, stride,
            total_patch, batch_size
        );
        cudaDeviceSynchronize();
    }

    if (gap > 0) {
        constant_ell<<<numblocks_high, threadsPerBlock>>>(
            d_input, d_output, d_saved_filter,
            output_channel, output_height, output_width,
            w_number_of_patch_input, kernel_height, kernel_width, stride,
            gap, total_patch, batch_size
        );
        cudaDeviceSynchronize();
    }

    if (low > 1024) {
        void *kernelArgs[] = {
            (void *)&d_ge_filter,
            (void *)&d_matrix,
            (void *)&low
        };
        cudaLaunchCooperativeKernel((void*)GE_over, numblocks, threadsPerBlock,
           kernelArgs);
    } else {
        GE_under<<<numblocks, threadsPerBlock>>>(d_ge_filter, d_matrix, low);
    }
    cudaDeviceSynchronize();

    align_output<<<numblocks_high, threadsPerBlock>>>(d_output, d_matrix, d_input,
       d_inversion_input,
        output_channel, output_height, output_width, kernel_height, kernel_width,
        stride, w_number_of_patch_input,
        input_channel, input_height, input_width,
        total_patch, low, high, gap, batch_size);
    cudaDeviceSynchronize();

    torch::Tensor data_tensor = torch::from_blob(
        d_inversion_input,
        {batch_size, input_channel, input_height, input_width},
        torch::TensorOptions().dtype(torch::kFloat32).device(torch::kCUDA)
    ).clone();

    cudaFree(d_inversion_input);
    cudaFree(d_saved_filter);
    cudaFree(d_ge_filter);
    cudaFree(d_matrix);

    return data_tensor;
}
```