DROPIT: DROPPING INTERMEDIATE TENSORS FOR MEMORY-EFFICIENT DNN TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

A standard hardware bottleneck when training deep neural networks is GPU memory. The bulk of memory is occupied by caching intermediate tensors for gradient computation in the backward pass. We propose a novel method to reduce this footprint - Dropping Intermediate Tensors (DropIT). DropIT drops min-k elements of the intermediate tensors and approximates gradients from the sparsified tensors in the backward pass. Theoretically, DropIT reduces noise on estimated gradients and therefore has a higher rate of convergence than vanilla-SGD. Experiments show that we can drop up to 90% of the intermediate tensor elements in fully-connected and convolutional layers while achieving higher testing accuracy for Visual Transformers and Convolutional Neural Networks on various tasks (*e.g.* classification, object detection). Our code and models are available at `https://anonymous.4open.science/r/dropit-iclr177submission`.

1 INTRODUCTION

The training of state-of-the-art deep neural networks (DNNs) (Krizhevsky et al., 2017; Simonyan & Zisserman, 2015; He et al., 2016; Vaswani et al., 2017; Dosovitskiy et al., 2021) for computer vision often requires a large GPU memory. For example, training a small Visual Transformer model ViT-B/16 (Dosovitskiy et al., 2021), with an input image of 224×224 and a standard batch size of 512, requires >80 GB GPU memory. Such a high memory requirement makes the training of DNNs out of reach for the average academic or practitioner without access to high-end GPU resources.

When training DNNs, the GPU memory has three primary uses (Chen et al., 2016; Sohoni et al., 2019): **model memory** for storing network parameters, **optimization memory** for buffering the gradients, and **tensor cache** for storing intermediate tensors (also called activations) for gradient computation. In many computer vision tasks, the training input is a large batch of images or videos. Therefore, previous studies (Feng & Huang, 2021; Gruslys et al., 2016; Chen et al., 2016; Rajbhandari et al., 2020) treat the tensor cache as the largest consumer of GPU memory.

For differentiable layers, standard implementations store the input tensors for computing the gradients during back-propagation. One option to reduce storage is to cache tensors from only some layers. Uncached tensors get recomputed on the fly during the backward pass – this is the strategy of gradient checkpointing (Feng & Huang, 2021; Gruslys et al., 2016; Chen et al., 2016; Bulo et al., 2018). Another option is to quantize the tensors after the forward computation and use the quantized values for gradient computation during the backward pass (Jain et al., 2018; Fu et al., 2020; Chakrabarti & Moseley, 2019; Evans & Aamodt, 2021; Liu et al., 2022) – this is known as activation compression training (ACT). Quantization can considerably reduce memory, but also brings inevitable performance drops. Accuracy drops can be mitigated by bounding the error at each layer through adaptive quantization (Evans & Aamodt, 2021; Liu et al., 2022), *i.e.* adaptive ACT. However, training time then suffers as extensive tensor profiling is necessary during training.

In this paper, we propose to sparsify activations by simply dropping elements from the intermediate tensor. We call our method Dropping Intermediate Tensors (DropIT). In the most basic setting, dropping indices can be selected randomly, though dropping based on a min-k ranking on the element magnitude is more effective. Both strategies are much simpler than the sensitivity checking and other profiling strategies, making DropIT much faster than adaptive ACT.

During training, the intermediate tensor is transformed over to a sparse format after the forward computation is complete. The sparse tensor is then recovered to general tensor during backward gradient computation with dropped indices filled with zero. Curiously, with the right dropping strategy and ratio, DropIT has improved convergence properties compared to SGD. We attribute this to the fact that DropIT can, theoretically, reduce noise on the gradients. In general, reducing noise will result in more precise and stable gradients. Experimentally, this strategy exhibits consistent performance improvements on various network architectures and different tasks.

To the best of our knowledge, we are the first to propose activation sparsification. The closest related line of existing work is ACT, but unlike ACT, DropIT leaves key elements untouched, which is crucial for ensuring accuracy. Nevertheless, DropIT is orthogonal to activation quantization, and the two can be combined for additional memory reduction with higher final accuracy. The key contributions of our work are summarized as follows:

- We propose DropIT, a novel strategy to reduce the activation memory by dropping the elements of the intermediate tensor.
- We theoretically and experimentally show that DropIT can be seen as a noise reduction on stochastic gradients which leads to better convergence.
- DropIT can work for various settings: training from scratch, fine-tuning on classification, object detection, etc. Our experiments demonstrate that DropIT can drop up to 90% of the intermediate tensor elements in fully-connected and convolutional layers with testing accuracy higher than the baseline for CNNs and ViTs. We also show DropIT is much better on accuracy and speed compared to SOTA activation quantization methods, and it can be combined with them to pursue higher memory efficiency.

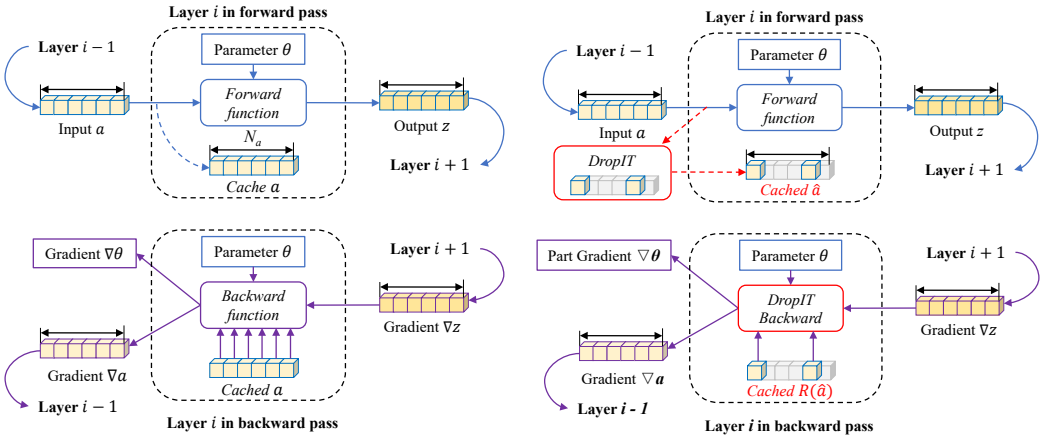
2 Related work

Memory-efficient training. Previous studies (Rajbhandari et al., 2020; Feng & Huang, 2021; Sohoni et al., 2019) have shown that during DNN training, GPU memory is allocated as model memory, optimization memory, and as a tensor cache. The model and optimization memory can be reduced with lightweight operations (Howard et al., 2017; Xie et al., 2017), distributed scheduling (Rajbhandari et al., 2020), and mixed precision training (Micikevicius et al., 2018). Yet for most vision models, the GPU memory is dominated by the tensor cache, as the intermediate tensor size is proportional to the input resolution and the batch size.

Gradient checkpointing. To reduce the tensor cache, gradient checkpointing (Chen et al., 2016; Gruslys et al., 2016; Feng & Huang, 2021) stores tensors from only a few layers and recomputes any uncached tensors when performing the backward pass; in the worst case scenario, this is equivalent to duplicating the forward pass, so any memory savings come as an extra computational expense. InPlace-ABN (Bulo et al., 2018) halves the tensor cache by merging batch normalization and activation into a single in-place operation. The tensor cache is compressed in the forward pass and recovered in the backward pass. Our method is distinct in that it does not require additional recomputation; instead, the cached tensors are sparsified heuristically.

Activation compression. (Jain et al., 2018; Fu et al., 2020; Chakrabarti & Moseley, 2019; Evans & Aamodt, 2021; Chen et al., 2021; Liu et al., 2022) explored lossy compression on the activation cache via low-precision quantization. (Wang et al., 2022) compressing high-frequency components while (Evans et al., 2020) adopts JPEG-style compression. In contrast to all of these methods, DropIT reduces activation storage via sparsification, which has been previously unexplored. In addition, DropIT is more lightweight than adaptive low-precision quantization methods (Evans & Aamodt, 2021; Liu et al., 2022).

Gradient approximation. Approximating gradients has been explored in large-scale distributed training to limit communication bandwidth for gradient exchange. (Strom, 2015; Dryden et al., 2016; Aji & Heafield, 2017; Lin et al., 2018) propose to drop gradients based on some fixed thresholds and send only the most significant entries of the stochastic gradients with the guaranteed convergence (Stich et al., 2018; Cheng et al., 2022; Chen et al., 2020). Instead of dropping gradient components, DropIT directly drops elements within intermediate tensors as our objective is to reduce the training memory.



(a) Baseline (b) DropIT

Figure 1: For a regular baseline network (a), the intermediate tensor is fully cached in the forward pass to be used for gradient computation during the backward pass. For DropIT, elements of the intermediate tensors are dropped during caching; only the retained elements are used for gradient computation during the backward pass. Best viewed in color.

3 METHODOLOGY

3.1 PRELIMINARIES

We denote the forward function and learnable parameters of i-th layer as l and θ, respectively. In the forward pass, l operates on the layer's input a to compute the output z: [1]

$$z = l(a, \theta). \tag{1}$$

For example, if layer i is a convolution layer, l would indicate a convolution operation with θ representing the kernel weights and bias parameter.

Given a loss function $F(\Theta)$, where Θ represents the parameters of the entire network, the gradient, with respect to θ at layer i, can be estimated according to the chain rule as

$$\nabla\theta \triangleq \frac{\partial F(\Theta)}{\partial \theta} = \nabla z \frac{\partial z}{\partial \theta} = \nabla z \frac{\partial l(a, \theta)}{\partial \theta}, \tag{2}$$

where $\nabla z \triangleq \frac{\partial F(\Theta)}{\partial z}$ is the gradient passed back from layer $i+1$. Note that the computation of $\frac{\partial l(a,\theta)}{\partial \theta}$ requires a if the forward function l involves tensor multiplication between a and θ. This is the case for many learnable layers, such as convolution and fully-connected layers. As such, a is necessary for estimating the gradient and is cached after it is computed during the forward pass, as illustrated in Figure 1(a). A common way to reduce storage for a is to store a quantized version (Jain et al., 2018; Fu et al., 2020; Chakrabarti & Moseley, 2019; Evans & Aamodt, 2021; Liu et al., 2022). Subsequent gradients in the backward pass are then computed using the quantized a.

The gradient ∇a can be estimated similarly via chain rule as

$$\nabla a \triangleq \frac{\partial F(\Theta)}{\partial a} = \nabla z \frac{\partial z}{\partial a} = \nabla z \frac{\partial l(a, \theta)}{\partial a}. \tag{3}$$

Analogous to Eq. 2, the partial $\frac{\partial l(a,\theta)}{\partial a}$ may depend on θ and θ is similarly stored in the model memory. Compared to a, however, θ typically occupies much less memory. This is especially true for convolution, as θ is shared with different input regions and batch samples. For example, ResNet-50 requires only ∼98MB GPU memory for storing parameters, but ∼18GB to cache intermediate

[1] Note that the output from the previous layer $i-1$, *i.e.* $a^i = z^{i-1}$. However, we assign different symbols to denote the input and output of a given layer explicitly; this redundant notation conveniently allows us, for clarity purposes, to drop explicit reference of the layer index i as a superscript.

Layer Type	Parameter θ	Tensor a
Convolution	$O(C_a C_z K^2)$	$O(BC_a HW)$
Fully Connected	$O(C_a C_z)$	$O(BC_a)$

Table 1: **Space complexity** for parameters and intermediate tensors in a single layer. B denotes batch size, C_a, C_z the number of input and output channels, K the kernel size, and H and W are height and width of the input tensor. Typically, $H \times W \gg K^2$, so the complexity for intermediate tensors is considerable with large B.

tensors when training on ImageNet with a standard batch size of 256. The space complexity for model parameters and intermediate tensors are compared in Table 1. The complexity for tensors becomes considerable if batch size B or feature map size $H \times W$ is large.

For some layers, *e.g.* batch normalization and activations, the partial $\frac{\partial l(a,\theta)}{\partial a}$ may also depend on a. In these cases, we do not drop the cache of intermediate tensors as this will affect subsequent back-propagation, causing an aggregated dropping effect on the gradient. For example, the gradient of the first layer will be affected by all the dropping errors and the convolution layers/fully connected layers, which will lead to sub-optimal convergence in the experiment. For DropIT, dropping happens only when the gradient flows to the parameters, which prevents the aggregation of errors from approximating the gradient.

3.2 Dropping intermediate tensors

Let $\mathcal{X}$ denote the set of all indices for an intermediate tensor a. Suppose that $\mathcal{X}$ is partitioned into two disjoint sets $\mathcal{X}_d$ and $\mathcal{X}_r$, *i.e.* $\mathcal{X}_r \cap \mathcal{X}_d = \emptyset$ and $\mathcal{X}_r \cup \mathcal{X}_d = \mathcal{X}$. In DropIT, we introduce a dropping operation $D(\cdot)$ to sparsify a into $\hat{a}$, where $\hat{a}$ consists of the elements $a_{\mathcal{X}_r}$ and the indices $\mathcal{X}_r$, *i.e.* $\hat{a} = D(a) = \{a_{\mathcal{X}_r}, \mathcal{X}_r\}$. The sparse $\hat{a}$ can be used as a substitute for a in Eq. 2. While sparsification can theoretically reduce both storage and computation time, we benefit only from storage savings in practice. We retain general matrix multiplication because the sparsity rate is insufficient for sparse matrix multiplication to provide meaningful computational gains. As such, the full intermediate tensors are recovered for gradient computation, *i.e.* $\nabla\theta \approx \nabla z \frac{\partial l(R(\hat{a}),\theta)}{\partial \theta}$, where $R(\cdot)$ represents the process that inflates $\hat{a}$ back to a general matrix with dropped indices filled with zero. The overall procedure is demonstrated in Figure 1(b).

Consider for a convolutional layer with C_z kernels of size $K \times K$. For the jth kernel, where $j \in [1, C_z]$, the gradient at location (u, v) for the kth channel is given by convolving incoming gradient ∇z and input a:

$$\nabla\theta_{j,k}(u,v) = \sum_{(n,x,y)\in\mathcal{X}} \nabla z_j^n(x,y) a_k^n(x',y'), \tag{4}$$

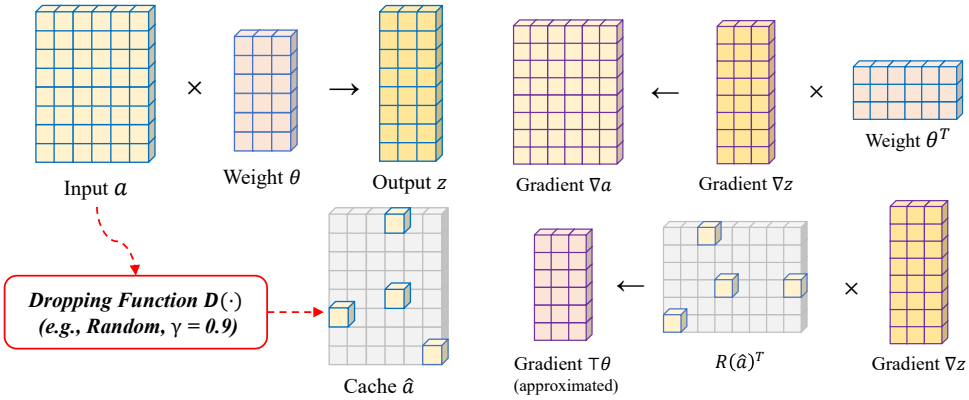
where $x' = x + u$ and $y' = y + v$. The set $\mathcal{X}$ in this case would denote the set of all sample indices $n \in [1, B]$ and all location indices $(x, y) \in [1, W] \times [1, H]$ in the feature map. Without any loss in generality, we can partition $\mathcal{X}$ into two disjoint sets $\mathcal{X}_r$ and $\mathcal{X}_d$ to split Eq. 4 as

$$\nabla\theta_{j,k}(u,v) = \Big[\sum_{(n,x,y)\in\mathcal{X}_d} \nabla z_j^n(x,y) a_k^n(x',y') + \underbrace{\sum_{(n,x,y)\in\mathcal{X}_r} \nabla z_j^n(x,y)\ a_k^n(x',y')}_{\top\theta_{j,k}(u,v)} \Big]. \tag{5}$$

Assume now, that some element $a_k^n(x', y')$ is small or near-zero; in CNNs and Transformers, such an assumption is reasonable due to preceding batch normalization and ReLU or GeLU activations (see Figure 3). Accordingly, this element's contribution to the gradient will also be correspondingly small. If we assign the spatial indices (x, y) in sample n of all small or near-zero elements to $\mathcal{X}_d$, then we can approximate the gradient $\nabla\theta_{j,k}(u, v)$ with simply the second term of Eq. 5. We denote the approximated gradient, as $g_{dropit} = \top\theta_{j,k}(u, v)$.

For a fully connected layer, the approximated gradient can be defined similarly as

$$g_{dropit} = \top\theta_{j,k} = \sum_{n\in\mathcal{X}_r} \nabla z_j^n\ a_k^n. \tag{6}$$



(a) Forward (b) Backward

Figure 2: Forward and backward of DropIT on the fully-connected layer (without bias). In the forward pass, we sparsify cache tensor and drop γ percentage storage. In the backward pass, only saved elements participate the gradient computation.

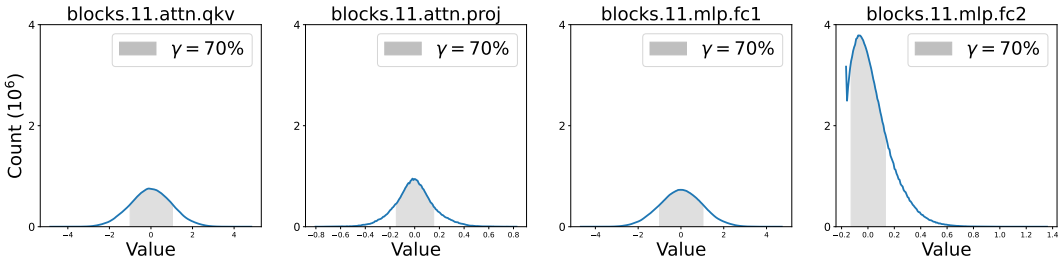


Figure 3: Distribution of element values in intermediate tensors' on DeiT-Ti. Dropped elements are shaded in grey. DropIT with min-k only discards elements that are close to zero. Here we only show the final block while observing that the distributions of other blocks are similar.

A visualization of the gradient approximation is shown in Figure 2. With the approximated gradient, we can use any standard deep learning optimization scheme to update the parameters.

3.3 Dropping Function $D(\cdot)$

We define the overall dropping rate as $\gamma = \frac{|\mathcal{X}_r|}{BC_a}$ for a fully connected layer and $\gamma = \frac{|\mathcal{X}_r|}{BC_aHW}$ for a convolutional layer. γ can be varied and will be used later to define the dropping function $D(\cdot)$. As we aim to drop elements with minimal contribution to the gradient, it is logical to perform a min-k based selection on the elements' magnitudes before dropping the elements. As a baseline comparison, we also select $\mathcal{X}_d$ based on uniform random sampling. We investigate the following options for $D(\cdot)$:

Random Elements: γ fraction of elements are dropped randomly within a mini-batch.

Min-K Elements: Within a mini-batch, we drop the smallest γ fraction of elements according to their absolute magnitudes.

3.4 Theoretical Analysis

Below, we analyze convergence for dropping min-k elements. The gradient of Stochastic Gradient Descent (SGD) is commonly viewed as Gradient Descent (GD) with noise:

$$g_{sgd} = g_{gd} + n(0, \xi^2), \tag{7}$$

where n represents some zero-mean noise distribution with a variance of ξ^2 introduced by variation in the input data batches.

With min-k dropping, the gradient becomes biased; we assume it can be modeled as:

$$g_{min\text{-}k} = \alpha g_{gd} + \beta n(0, \xi^2). \tag{8}$$

That is, min-k dropping results in a bias factor α while affecting noise by a factor of β. α and β varies each iteration, i.e., $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_t\}$ and $\beta = \{\beta_1, \beta_2, ..., \beta_t\}$. Additionally in Appendix A.2, we provide a nonlinear approximation to g_{min-k} which achieves same convergence.

By scaling the learning rate with a factor of $\frac{1}{\alpha}$, the gradient after min-k dropping as given in Eq. 8 can also be expressed as:

$$g_{min\text{-}k} = g_{gd} + \frac{\beta}{\alpha} n(0, \xi^2). \tag{9}$$

We can formally show (see Appendix A.3) that $\mathbb{E}[\alpha] \geq \mathbb{E}[\beta] \geq 1 - \gamma$ and therefore $\mathbb{E}[\frac{\beta}{\alpha}] \leq 1$. This suggests min-$k$ dropping reduces the noise of the gradient. With less noise, better theoretical convergence is expected.

Similar to convergence proofs in most optimizers, we will assume that the loss function F is L-smooth. Under the L-smooth assumption, for SGD with a learning rate η and min-k dropping with a learning rate $\frac{\eta}{\alpha_t}$, we can reach the following convergence after T iterations:

$$\text{SGD:} \quad \frac{1}{T}\mathbb{E}\sum_{t=1}^{T} \|\nabla F(x_t)\|^2 \leq \frac{2(F(x_1) - F(x^*))}{T\eta} + \eta L \xi^2 \tag{10}$$

$$\text{DropIT with min-}k\text{:} \quad \frac{1}{T}\mathbb{E}\sum_{t=1}^{T} \|\nabla F(x_t)\|^2 \leq \frac{2(F(x_1) - F(x^*))}{T\eta} + \eta L \xi^2 \frac{1}{T}\sum_{t=1}^{T} \frac{\beta_t^2}{\alpha_t^2}, \tag{11}$$

where x^* indicates an optimal solution. Full proof can be found in Appendix A.1. Note that the two inequalities differs only by the second term in the right-hand side. α_t represents the bias caused by dropping at t-th iteration and β_t measures the noise reduction effect after dropping. We further investigate α and β in supplementary and show that under certain conditions, $\mathbb{E}[\alpha] \geq \mathbb{E}[\beta]$, thereby reducing the noise and improving the convergence of DropIT from standard SGD.

3.5 DropIT for Networks

The material above has discussed dropping tensor elements from the cache of a single layer. DropIT is theoretically applicable for all convolutional and fully-connected layers in a network since it does not affect the forward pass. For Visual Transformers (Dosovitskiy et al., 2021), we apply DropIT for most learnable layers, though we ignore the normalization and activations like LayerNorm (Ba et al., 2016) and GELU (Hendrycks & Gimpel, 2016)). The applicable layers include fully-connected layers in multi-head attention and MLPs in each block, the beginning convolutional layer (for patches projection), and the final fully-connected classification layer. For CNNs the applicable layers includes all convolutional layers and the final fully-connected classification layer. We leave networks unchanged during inference.

4 Experiments

Experimentally, we show that DropIT achieves lower training loss and higher testing accuracy when training from scratch on ImageNet-1k (Russakovsky et al., 2015). DropIT is also applicable for transfer learning tasks, including finetuning on CIFAR-100 (Krizhevsky et al., 2009) and object detection and instance segmentation on MS-COCO (Lin et al., 2014). Finally, we compare and combine DropIT with the recent SOTA activation compression methods (Pan et al., 2022; Liu et al., 2022), to highlight our advantages in terms of accuracy, speed, and memory.

4.1 Experimental details

Models. For image classification experiments, we use the Image Transformer (DeiT) (Touvron et al., 2021). DeiT is preferable to the vanilla ViT (Dosovitskiy et al., 2021) because DeiT provides results

Strategy	Dropping Rate γ									
	$0\%(Baseline)$	10%	20%	30%	40%	50%	60%	70%	80%	90%
Random	72.1*	**72.4**	**72.4**	72.0	71.7	70.8	69.6	68.1	65.8	60.8
Min-K	72.1*	**72.1**	**72.1**	**72.2**	**72.4**	**72.5**	**72.4**	**72.1**	70.8	66.4

From Touvron et al. (2021)'s official implementation, we obtain 72.13 with public weights and our training.

Table 2: Ablation study on dropping strategy and dropping rate. Reported results are top-1 accuracy on ImageNet-1k validation set, achieved by DeiT-Ti training from scratch on ImageNet-1k training set. We highlight the accuracy higher than baseline ($\geq$72.1).

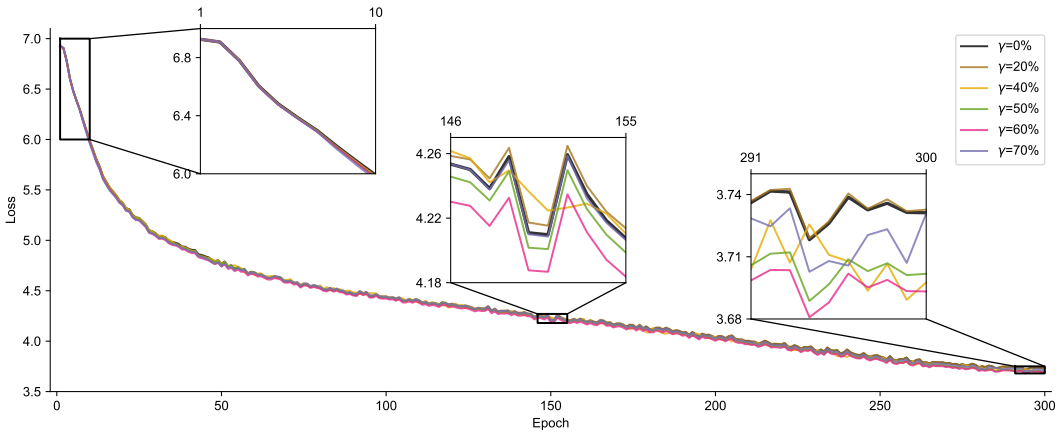


Figure 4: Training loss curves of Min-K DropIT. Baseline ($\gamma = 0\%$) is bolded. $\gamma = 80\%, 90\%$ are hidden as their losses are obviously higher than the baseline. $\gamma = 10\%, 30\%$ are also hidden for easier viewing. $\gamma = 40\%{\sim}70\%$ achieve lower loss than baseline at the end. Best viewed in color.

on ImageNet-1k training from scratch, while ViT requires finetuning from ImageNet-21k. Note that DeiT and ViT has the same architecture and only differ in their training hyper-parameters (*e.g.* batch size, learning rate, weight decay). For transfer learning experiments, we select the widely-used Faster/Mask R-CNN models (Ren et al., 2017; He et al., 2017) and validate our method for object detection and instance segmentation.

Implementation Details. We follow the official implementations of DeiT without distillation and Faster/Mask R-CNN[2], keeping *all* hyper-parameters the same; the only difference is the computation of gradients with DropIT. Our implementation is done with PyTorch 1.12 (Paszke et al., 2019) and we use the `torch.autograd`. We reimplement conv/fc layers so that we can sparsify tensor cache (dense $\rightarrow$ coordinate format) after the forward pass, and recover the sparsified tensor (coordinate format $\rightarrow$ dense) during the backward pass. With PyTorch native functions, our framework does not introduce/reduce extra overhead and guarantees a fair comparisons. DropIT's min-*k* strategy is implemented by `torch.topk`, where elements of $1-\gamma$ proportion with largest absolute value are maintained, with the corresponding indices of these elements. For all our experiments, we follow DeiT (Touvron et al., 2021) to use a fixed random seed 0. Training speed and memory are measured on NVIDIA RTX A5000 GPUs. See Appendix A.9 for more details.

4.2 Impact on Accuracy

Training from scratch on ImageNet-1k. Table 2 shows that training DeiT-Ti from scratch without DropIT (baseline) has a top-1 accuracy of 72.1 on ImageNet-1k. Random dropping matches or improves the accuracy (72.4) when $\gamma \leq 20\%$, but with higher γ ($\gamma \geq 30\%$), accuracy progressively decreases fm the baseline. The phenomenon can be explained by: (1) Small amounts of random dropping ($\gamma \leq 20\%$) can be regarded as adding random noise to the gradient. The noise has a regularization effect on the network optimization to improve the accuracy, similar to what is observed in previous studies (Neelakantan et al., 2015; Evans & Aamodt, 2021). (2) Too much random dropping

[2] `github.com/facebookresearch/deit`, `pytorch.org/vision/main/models/`

(a) Image classification finetuning on CIFAR-100. DeiT networks are initialized from their public pretrained ImageNet-1k weights.

Network	DropIT	Top-1 Accuracy
DeiT-S	-	89.7
	$\gamma = 90\%$	**90.1**
DeiT-B	-	90.8
	$\gamma = 90\%$	**91.3**

(b) Detection & instance segmentation on COCO. *R50-FPN* denotes ResNet-50 with FPN (Lin et al., 2017), initialized from public ImageNet-1k weights.

Network	DropIT	AP^{box}	AP^{mask}
Faster R-CNN (R50-FPN)	-	37.0	n/a
	$\gamma = 90\%$	**37.2**	n/a
Mask R-CNN (R50-FPN)	-	37.9	34.5
	$\gamma = 80\%$	**38.5**	34.5

Table 3: Finetuning for downstream application: CIFAR-100 classification, COCO instance. , initialized from their public pretrained ImageNet-1k weights. segmentation.

Cached	Dropping Rate γ				
	$\gamma = 0\%$	$\gamma = 60\%$	$\gamma = 70\%$	$\gamma = 80\%$	$\gamma = 90\%$
Tensor	11.26 G	4.50 G (-60%)	3.38 G (-70%)	2.25 G (-80%)	1.13 G (-90%)

Table 4: Memory cost of DropIT cached tensors (without indices) in different γ. DropIT can precisely reduce the memory according to γ. The measured model is DeiT-S with a batch size of 1024.

($\gamma \geq 30\%$) results in deviations which can no longer be seen as small gradient noise, hence reducing performance.

With min-*k* dropping, DropIT can match or exceed the baseline accuracy over a wide range of γ ($\leq 70\%$). Intuitively, training from scratch should be difficult with DropIT, especially under large dropping rates, as the computed gradients are approximations. However, our experiments demonstrate that DropIT achieves 0.4% and 0.3% higher accuracy in $\gamma = 50\%$ and 60%, respectively. In fact, DropIT can match the baseline accuracy even after discarding 70% of the elements.

Fig. 4 compares the loss curves when training from scratch on the baseline DeiT-Ti model without and with DropIT using a min-*k* strategy. The loss curves of DropIT with various γ value follow the same trend as the baseline; up to some value of γ, the curves are also but are consistently lower than the baseline, with $\gamma = 50\%, 60\%$ achieving the lowest losses and highest accuracies. As such, we conclude that DropIT accurately approximates the gradient while reducing noise as per our theoretical analysis.

Fine-tuning on CIFAR-100. Table 3(a) shows that DeiT networks can be fine-tuned with DropIT to achieve higher than baseline accuracies even while dropping up to 90% intermediate elements. Compared to training from scratch from Table 2, DropIT can work with a more extreme dropping rate (90% vs. 70%). We interpret that this is because the network already has a good initialization before fine-tuning, thereby simplifying the optimization and allowing a higher γ to be used.

Backbone fine-tuning, head training from scratch on COCO. We have investigated DropIT in two settings: training from scratch, fine-tuning from an pre-trained network. We continue to study backbone network initialized (*e.g.* ImageNet pre-training) but others (*e.g.* RPN, R-CNN head) not initialized, which is the common practice in object detection. As shown in Table 3(b), DropIT can steadily improve the detection accuracy (AP^{box}). When $\gamma = 80\%$, we observe that DropIT yields an impressive 0.6 AP gain in Mask R-CNN. However, such a gain is not observed on AP^{mask}. We think the segmentation performance may be highly related to the deconvolutional layers in mask head, but they are not supported by DropIT currently. We will investigate this problem in future work. These experiments also demonstrate the effectiveness for DropIT on CNNs. In Appendix A.4, we also show that DropIT is effective for ResNet training from scratch on ImageNet-1k.

4.3 Impact on Memory & Speed, SOTA Comparison

Activation Reduction. Table 4 shows the activation reduction achieved by DropIT. In DropIT applied layers (FC layers of DeiT-S), the total reserved activation (batch size $= 1024$) is 11.26 G. When we use DropIT to discard activations, the memory reduction is precisely controlled by γ, *i.e.* $\gamma = 90\%$ means the reduction is 11.26×0.9. DropIT does incur some memory cost for indexing, but as we show next, the maximum GPU memory can still be reduced.

Benchmark	FC Cache	Others Cache	Acc	MaxM	MaxM (-Index)	Speed (ms)
DeiT-S on CIFAR-100	- (Baseline)	- (Baseline)	89.7	6.66 G	6.66 G	**172**
	DropIT ($\gamma = 90\%$)	-	**90.1**	**5.59 G**	**5.29 G**	212
	MESA (8-bit)	MESA (8-bit)	89.7	3.52 G	3.52 G	416
	DropIT($\gamma = 90\%$)	MESA (8-bit)	**89.9**	**3.27 G**	**2.97 G**	**375**
	GACT (4-bit)	GACT (4-bit)	89.7	2.16 G	2.16 G	290+49*
	DropIT($\gamma = 90\%$)	GACT (4-bit)	**90.0**	2.27 G	**1.97 G**	**286+25***

**Note: GACT has a time-consuming sensitivity profiling computation every 1000 iterations. It costs 49.81 and 25.43 (+DropIT) seconds in our benchmark. So we add an average of this time over 1000 iterations).*

Table 5: Compare and combine with state-of-the-art ACT methods. FC: fully-connected. MaxM: maximum memory. MaxM (- Index): maximum memory without index (moved to CPU). We follow MESA to use batch size 128 to measure memory and speed on a single GPU.

Comparison and Combination with SOTA. In Table 5, we compare and combine DropIT with state-of-the-art activation quantization methods. Measuring performance individually, with $\gamma = 90\%$, DropIT improves accuracy by 0.4 and reduces maximum memory by 1.07 G (1.37G activations - 0.3G indexing), and slightly increases the time (40 ms) per iteration. The max memory reduction is less than that shown Table 4 because activations from non-applicable layers still occupy considerable memory. Therefore, a natural idea to supplement DropIT is to perform activation quantization for layers without DropIT. We next present combination results of DropIT with recent methods MESA (Pan et al., 2022) and GACT (Liu et al., 2022).

As shown in Table 5, MESA can reduce memory with 8-bit quantization and has no impact on the baseline accuracy (89.7). However, the time cost of MESA algorithm is also considerable, and is $416/172 \approx 2.4\times$ slower than baseline and $416/212 \approx 2\times$ more than DropIT, with no accuracy improvement in CIFAR-100 finetuning. MESA achieves 71.9 accuracy of DropIT-Ti on ImageNet-1k, but DropIT can go up to 72.5 (Table 2, $\gamma = 50\%$). We can combine MESA with DropIT by applying DropIT in the `conv`/`fc` layers and applying MESA in the other layers. Together, the accuracy, memory, and speed are all improved over MESA alone, conclusively demonstrating the effectiveness of DropIT.

We compare similarly to GACT; Table 5 shows that at 4-bits, it can reduce max-memory even further. Combining GACT with DropIT marginally increase the max-memory due to DropIT's indexing consumption; however, tehre are both accuracy and speed gains. Furthermore, GACT reports 0.2$\sim$0.4 AP^{box} loss on COCO (Liu et al., 2022), though our DropIT can produce 0.6 AP^{box} improvement on COCO (Table 3(b)). To sum up, DropIT has its unique advantages on accuracy and speed compared to existing activation quantization methods. Although it saves less memory than the latter, we can combine the two to achieve higher memory efficiency.

5 Conclusion

In this paper, we proposed the Dropping Intermediate Tensors (DropIT) method to reduce the GPU memory cost during the training of DNNs. Specifically, DropIT drops elements in intermediate tensors for the memory-efficient tensor cache, and it recovers sparsified tensors from the remaining elements in the backward pass to compute the gradient. Our experiments showed that DropIT can improve the accuracies of DNNs and save up to 27% of GPU memory on different backbones and datasets. DropIT provides a new perspective to reduce the GPU memory cost during DNN training. For future work, DropIT can be explored in more downstream tasks such as action recognition and graph neural networks.

References

Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In *EMNLP*, pp. 440–445, 2017.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv: 1607.06450*, 2016.

Samuel Rota Bulo, Lorenzo Porzi, and Peter Kontschieder. In-place activated batchnorm for memory-optimized training of dnns. In *CVPR*, pp. 5639–5647, 2018.

Ayan Chakrabarti and Benjamin Moseley. Backprop with approximate activations for memory-efficient network training. In *NeurIPS*, pp. 2426–2435, 2019.

Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W. Mahoney, and Joseph Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *ICML*, volume 139, pp. 1803–1813, 2021.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv*, 1604.06174, 2016.

Zhao Chen, Jiquan Ngiam, Yanping Huang, Thang Luong, Henrik Kretzschmar, Yuning Chai, and Dragomir Anguelov. Just pick a sign: Optimizing deep multitask models with gradient sign dropout. *NeurIPS*, pp. 2039–2050, 2020.

Feng Cheng, Mingze Xu, Yuanjun Xiong, Hao Chen, Xinyu Li, Wei Li, and Wei Xia. Stochastic backpropagation: A memory efficient strategy for training video models. In *CVPR*, pp. 8301–8310, 2022.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *MLHPC workshop*, pp. 1–8, 2016.

R. David Evans and Tor M. Aamodt. AC-GC: lossy activation compression with guaranteed convergence. In *NeurIPS*, pp. 27434–27448, 2021.

R David Evans, Lufei Liu, and Tor M Aamodt. Jpeg-act: accelerating deep learning via transform-based lossy compression. In *ISCA*, pp. 860–873, 2020.

Jianwei Feng and Dong Huang. Optimal gradient checkpoint search for arbitrary computation graphs. In *CVPR*, pp. 11433–11442, 2021.

Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. In *ICML*, pp. 3304–3314, 2020.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *NeurIPS*, pp. 4125–4133, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pp. 770–778, 2016.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. In *ICCV*, pp. 2980–2988, 2017.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arxiv:1606.08415*, 2016.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, pp. 776–789, 2018.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *ECCV*, pp. 740–755, 2014.

Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. In *CVPR*, pp. 936–944, 2017.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *ICLR*, 2018.

Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, Michael Mahoney, and Alvin Cheung. GACT: Activation compressed training for generic network architectures. In *ICML*, pp. 14139–14152, 2022.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *ICLR*, 2018.

Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv:1511.06807*, 2015.

Zizheng Pan, Peng Chen, Haoyu He, Jing Liu, Jianfei Cai, and Bohan Zhuang. Mesa: A memory-saving training framework for transformers. *arXiv:2111.11124*, 2022.

Adam Paszke, Sam Gross, and Francisco et al Massa. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8026–8037, 2019.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *SC*, pp. 20, 2020.

Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6): 1137–1149, 2017.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv:1904.10631*, 2019.

Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with memory. In *NeurIPS*, pp. 4452–4463, 2018.

Nikko Strom. Scalable distributed DNN training using commodity GPU cloud computing. In *Interspeech*, pp. 1488–1492, 2015.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *ICML*, pp. 10347–10357, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, pp. 6000–6010, 2017.

Guanchu Wang, Zirui Liu, Zhimeng Jiang, Ninghao Liu, Na Zou, and Xia Hu. A concise framework of memory efficient training via dual activation precision. *arXiv:2208.04187*, 2022.

Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *CVPR*, pp. 5987–5995, 2017.

A APPENDIX

A.1 COMPLETE CONVERGENCE ANALYSIS

Here we prove convergence of DropIT with min-*k* dropping strategy. By scaling the learning rate with a factor of $\frac{1}{\alpha}$, the gradient of min-*k* dropping is modeled as:

$$g_{min\text{-}k} = g_{gd} + \frac{\beta}{\alpha} n(0, \xi^2). \tag{12}$$

where n is zero-mean noise with a variance of ξ^2, and α, β are varied each iteration.

We assume that the loss function F is L-smooth, i.e., F is differentiable and there exists a constant $L > 0$ such that

$$F(y) \leq F(x) + \langle \nabla F(x), y - x \rangle + \frac{L}{2}\|y - x\|^2, \qquad \forall x, y \in \mathbb{R}^d. \tag{13}$$

Performing Taylor expansion we have:

$$\begin{aligned} \mathbb{E}[F(x_{t+1})] &\leq F(x_t) - \langle \nabla F(x_t), x_{t+1} - x_t \rangle + \frac{\eta^2 L}{2} \mathbb{E}[\|\nabla F(x_t)\|^2] \\ &\leq F(x_t) - \eta \|\nabla F(x_t)\|^2 + \frac{\eta^2 L \xi^2 \beta_t^2}{2\alpha_t^2} \end{aligned} \tag{14}$$

Rearranging the terms of the above inequality and dividing by $\frac{\eta}{2}$, we obtain:

$$\|\nabla F(x_t)\|^2 \leq \frac{2(F(x_t) - \mathbb{E}[F(x_{t+1})])}{\eta} + \frac{\eta L \xi^2 \beta_t^2}{\alpha_t^2} \tag{15}$$

Summing up from $t = 1$ to T and divided by T, we get:

$$\frac{1}{T}\mathbb{E}\sum_{t=1}^{T} \|\nabla F(x_t)\|^2 \leq \frac{2(F(x_1) - F(x^*))}{T\eta} + \eta L \xi^2 \frac{1}{T}\sum_{t=1}^{T} \frac{\beta_t^2}{\alpha_t^2} \tag{16}$$

where x^* indicates an optimal solution.

A.2 MODELING MIN-*k* DROPPING GRADIENT WITH NONLINEAR FUNCTION

We can replace Eq. 8 (gradient model of min-*k* dropping) with nonlinear function and still achieve the same convergence as in Eq. 10.

The gradient is biased with min-*k* dropping, we assume it can be modeled as:

$$g_{min\text{-}k} = g_{gd} + \beta n(0, \xi^2) + b, \tag{17}$$

where b is a bias and $||b||^2 \leq (1-\alpha)||g_{gd}||^2$. α and β varies each iteration, i.e., $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_t\}$ and $\beta = \{\beta_1, \beta_2, ..., \beta_t\}$.

Assuming the loss function F is L-smooth, we obtain:

$$\begin{aligned}
\mathbb{E}F(x_{t+1}) &\leq F(x_t) - \langle \nabla F(x_t), x_{t+1} - x_t \rangle + \frac{\eta^2 L}{2}\mathbb{E}||\nabla F(x_t) + b||^2 \\
&= f(x_t) - \eta\langle \nabla F(x_t), \nabla F(x_t) + b\rangle + \frac{\eta^2 L \xi^2 \beta_t^2}{2} \\
&\leq f(x_t) - \eta\langle \nabla F(x_t), \nabla F(x_t) + b\rangle + \frac{\eta}{2}||\nabla F(x_t) + b||^2 + \frac{\eta^2 L \xi^2 \beta_t^2}{2} \\
&= f(x_t) - \frac{\eta}{2}\Big(2\langle \nabla F(x_t), \nabla F(x_t) + b\rangle - ||\nabla F(x_t) + b||^2\Big) + \frac{\eta^2 L \xi^2 \beta_t^2}{2} \\
&= f(x_t) - \frac{\eta}{2}\Big(||\nabla F(x_t)||^2 - ||b||^2\Big) + \frac{\eta^2 L \xi^2 \beta_t^2}{2} \\
&\leq f(x_t) - \frac{\eta\alpha_t}{2}||\nabla F(x_t)||^2 + \frac{\eta^2 L \xi^2 \beta_t^2}{2}
\end{aligned} \tag{18}$$

Rearranging the terms of the above inequality and dividing by $\frac{\eta\alpha_t}{2}$, we obtain:

$$\|\nabla F(x_t)\|^2 \leq \frac{2(F(x_t) - \mathbb{E}[F(x_{t+1})])}{\eta\alpha_t} + \frac{\eta L \xi^2 \beta_t^2}{\alpha_t} \tag{19}$$

Using a learning rate of $\frac{\eta}{\alpha_t}$ instead, we have:

$$\|\nabla F(x_t)\|^2 \leq \frac{2(F(x_t) - \mathbb{E}[F(x_{t+1})])}{\eta} + \frac{\eta L \xi^2 \beta_t^2}{\alpha_t^2} \tag{20}$$

Summing up from $t = 1$ to T and divided by T, we get:

$$\frac{1}{T}\mathbb{E}\sum_{t=1}^{T}\|\nabla F(x_t)\|^2 \leq \frac{2(F(x_1) - F(x^*))}{T\eta} + \eta L \xi^2 \frac{1}{T}\sum_{t=1}^{T}\frac{\beta_t^2}{\alpha_t^2} \tag{21}$$

where x^* indicates an optimal solution. The convergence is exactly the same as in Appendix A.1.

A.3 Theoretical analysis on α and β

In this section we compare the gradients of SGD and DropIT with min-*k* dropping. Note we slightly change the notation of the gradients from g_{sgd} and g_{mink} in the main paper to improve clarity for element-wise analysis. We denote the gradient of SGD as G and DropIT as G'. Both gradients are computed by an input tensor A and intermediate tensor Z. In DropIT we drop γ percent of the elements in A. Thus we have:

$$G = A \times Z \tag{22}$$

$$G' = (A \odot D) \times Z, \tag{23}$$

where $\odot$ is element-wise multiplication and D is a dropping mask where each element is either 1 or 0.

From an element-wise viewpoint, we rewrite the computation of gradients:

$$g_{ij} = \sum_k a_{ik} z_{kj} \tag{24}$$

$$g'_{ij} = \sum_k a_{ik} d_{ik} z_{kj}, \tag{25}$$

where d_{ik} is a mask, i.e., d_{ik} is either 0 or 1 depending on a_{ik}.

For the simplicity of analysis, we assume A and Z are independent. Let μ be the mean value of A and c be the mean of dropped elements, after dropping, $A \odot D$ has a mean of $\mu - c$. Taking expectation over all possible inputs, we have:

$$\begin{aligned}\mathbb{E}[g'_{ij}] &= \mathbb{E}[\alpha g_{ij}] \\ &= \frac{\mu - c}{\mu} \sum_k a_{ik} z_{kj} \\ &= \frac{\mu - c}{\mu} g_{ij}. \end{aligned} \tag{26}$$

Therefore the bias caused by dropping is expected to be $\frac{\mu - c}{\mu}$. Assuming the mean value of A is μ and the mean of dropped value is c, after dropping, $D(A)$ has a mean of $\mu - c$. Thus the bias caused by dropping is $\mathbb{E}[\alpha] = \frac{\mu - c}{\mu}$. Recall that we drop elements with small absolute values. In the extreme case where every element in A has the same value as μ, c will reach the upper bound $\gamma\mu$. Therefore, $\mathbb{E}[\alpha] \geq 1 - \gamma$.

Now we analyze on noise and compute β. Due to the variation on input samples, we have noise in A and Z, which results in noise in G and G'. To highlight the noise, we rewrite a noisy element x as $\bar{x} + n_x$, where $\bar{x}$ is the mean value of x and n_x is a zero-mean noise. Applying it to Eq.24 and Eq.25 we arrive at:

$$\bar{g}_{ij} + n_g = \sum_k (\bar{a}_{ik} + n_a)(\bar{z}_{kj} + n_z) \tag{27}$$

$$\bar{g'}_{ij} + n_{g'} = \sum_k d_{ik}(\bar{a}_{ik} + n_a)(\bar{z}_{kj} + n_z). \tag{28}$$

Focusing on the noise of gradients, we obtain:

$$n_g = \sum_k (\bar{a}_{ik} n_z + \bar{z}_{kj} n_a + n_a n_z) \tag{29}$$

$$n_{g'} = \sum_k (d_{ik} \bar{a}_{ik} n_z + d_{ik} \bar{z}_{kj} n_a + d_{ik} n_a n_z). \tag{30}$$

Recall that d_{ik} is a mask depending on a_{ik} and therefore depending on $\bar{a}_{ik}$, thus from Eq.26 we have

$$\mathbb{E}[d_{ik} \bar{a}_{ik}] = \frac{\mu - c}{\mu} \bar{a}_{ik} \approx \bar{a}_{ik} > (1 - \gamma) \bar{a}_{ik}. \tag{31}$$

Because γ percent of D is 0 and the other $1 - \gamma$ percent of D is 1, we obtain $\mathbb{E}[d_{ik}] = 1 - \gamma$. Plugging them in Eq.30 we have:

$$\begin{aligned}\mathbb{E}[n_{g'}] &= \mathbb{E}[\beta n_g] \\ &= \sum_k (\frac{\mu - c}{\mu} \bar{a}_{ik} n_z + (1 - \gamma) \bar{z}_{kj} n_a + (1 - \gamma) n_a n_z) \\ &\leq \frac{\mu - c}{\mu} \sum_k (\bar{a}_{ik} n_z + \bar{z}_{kj} n_a + n_a n_z) \\ &= \mathbb{E}[\alpha n_g], \end{aligned} \tag{32}$$

where the inequality is satisfied due to $c \leq \gamma\mu$.

This result tells us $\mathbb{E}[\beta] \leq \mathbb{E}[\alpha]$, which suggests DropIT with min-*k* dropping has a noise reduction effect and should converge better than SGD.

Network	Method	Top-1
ResNet-50	-	76.1
	DropIT ($\gamma = 70\%$)	**76.3**

Table 6: Results of ResNet-50 with and without DropIT, training from scratch on ImageNet-1k.

Dataset	Method	Top-1	Top-5	Memory (GB)
CIFAR-100	ResNet-18 (32×32)	77.96	94.05	648
	ResNet-18 (32×32) + DropIT ($\gamma = 0.8$)	**78.17**	**94.19**	**598**
	ViT-B/16 (224×224)	90.32	98.88	20290$\times$4
	ViT-B/16 (224×224) + DropIT ($\gamma = 0.9$)	**90.90**	**99.02**	**16052**$\times$4
ImageNet	ResNet-18 (224×224)	69.76	89.08	2826
	ResNet-18 (224×224) + DropIT ($\gamma = 0.8$)	**69.85**	**89.39**	**2600**
	ViT-B/16 (224×224)	83.40	96.96	20290$\times$4
	ViT-B/16 (224×224) + DropIT ($\gamma = 0.9$)	**83.61**	**97.01**	**16056**$\times$4

Table 7: More results of different network architecture achieved by DropIT. ResNet-18 results are training from scratch, and ViT-B/16 are fine-tuning from public ImageNet-21k weights.

A.4 ResNet-50 training from scratch on ImageNet-1k

We follow `torchvision` training script [3] to train ResNet with and without DropIT. No training hyper-parameters are changed. See Table 6, similar to DeiT, DropIT also achieves higher accuracy on ResNet, which further demonstrates the generalization of DropIT.

A.5 More network results

We present more results of different network architectures as shown in Table 7. ResNet-18 are trained from scratch by 90 epochs on ImageNet, totally following `torchvision` reference code. ViT-B/16 is fine-tuned in 3 epochs from its ImageNet-21k pretrained weights. Our proposed DropIT can improve the accuracy for these setting with lower GPU memory cost.

A.6 A simplified implementation example for DropIT

```python
from torch import nn, Tensor
from torch.autograd import Function

class DropITLinear(Function):
    @staticmethod
    def forward(ctx, x:Tensor, weight:nn.Parameter, bias:nn.Parameter):
        # select_topk is a simplified function
        x_dropped, idxs = select_topk(x)
        ctx.save_for_backward(x_dropped, idxs, weight)

        # linear_fackward is a simplified function
        return linear_forward(x, weight, bias)

    @staticmethod
    def backward(ctx, grad_output:Tensor):
        x_dropped, idxs, weight = ctx.saved_tensors

        # recover_topk is a simplified function
        x = recover_topk(x_dropped, idxs)

        # linear_backward is a simplified function
        grad_input, grad_weight, grad_bias = linear_backward(x, weight)
        return grad_input, grad_weight, grad_bias
```

[3] https://github.com/pytorch/vision/tree/main/references/classification#resnet

```

def dropit_linear_forward(self, x):
    return DropITLinear.apply(x, self.weight, self.bias)

# usage: replace forward function of PyTorch Linear layer
from functools import partial
fc = nn.Linear(..., ...)
fc.forward = partial(dropit_linear_forward, fc)
```

A.7 WHY DROPIT IS NOT USED FOR THE NETWORK FIRST & FINAL LAYERS

We do not apply DropIT to `conv/fc` layer if it is the first/final layer of the network. The reason is that this does not save memory:

(1) The first layer: The logic of our DropIT on saving memory can be concluded as: creating a smaller tensor $\mathtt{x}_{dropped}$ (*i.e.* by `torch.topk`) from input tensor `x`, then the input tensor `x` will be automatically released by python garbage collection. However, popular code style is like:

```
dataloader = ...
loss_func = ...

def model(x):
    x = layer1(x)
    x = layer2(x) # x can be released with DropIT
    ...
    x = layeri1(x) # x can be released with DropIT
    x = layeri(x) # x can be released with DropIT, but cannot save
    maximum memory
    return x

for x, y in dataloader:
    x = model(x) # x will not be released with DropIT
    loss_func(x, y).backward()
    ...
```

As we can see, in the `dataloader` loop, the input `x` to `model` can only be recycled when `model` running is finished. So, using DropIT in `layer1` will not reduce maximum memory — instead, it will increase the maximum memory as DropIT created a new $\mathtt{x}_{dropped}$.

(2) The final layer: it is easy to understand that DropIT using in the final layer has no effect on memory. See the code block, when running to `layeri`, the maximum memory should be `layer1` $\sim$ `layeri1` cached tensors plus `x` input to `layeri`. If we use DropIT at `layeri`, then there would be an extra $\mathtt{x}_{dropped}$ produced, making the maximum memory even higher.

A.8 HOW TO SELECT γ OF DROPIT

From our experiments, we recommend $\gamma = 70\%$ for training from scratch and $\gamma = 80, 90\%$ for fine-tuning. As DropIT incurs memory cost for indexing, γ should be larger than 50% to be meaningful (assuming index data type is `int32` with the same number of bits of `float32` for activation). Empirically, we observe that γ is reflected consistently in both training loss and testing accuracy. A too-high γ which will bias the gradient will have training losses higher than the baseline. As such, an alternative way to select γ is to observe the training loss after a some iterations (*e.g.* 100); if it is lower than the baseline, then the testing accuracy is likely to improve as well.

A.9 MORE EXPERIMENTAL DETAILS

We list the detailed key training hyper-parameters, though they are totally the same with the offical implementations:

• *DeiT-Ti, training from scratch, ImageNet-1k, w/wo DropIT*[4]: batch size 1024, AdamW optimizer, learning rate 10^{-3}, weight decay 0.05, cosine LR schedule, 300 epochs, with auto mixed precision (AMP) training;

• *DeiT-S, finetuning from official DeiT-S ImageNet-1k weights, CIFAR-100, w/wo DropIT*[5]: batch size 768, SGD optimizer (momentum 0.9), learning rate 10^{-2}, weight decay 10^{-4}, cosine LR schedule, 1000 epochs, with AMP training;

• *DeiT-B, finetuning from official DeiT-B ImageNet-1k weights, CIFAR-100, w/wo DropIT*[6]: batch size 768, SGD optimizer (momentum 0.9), learning rate 10^{-2}, weight decay 10^{-4}, cosine LR schedule, 1000 epochs, with AMP training;

• *Faster R-CNN, finetuning from* `torchvision` *ResNet-50 ImageNet-1k weights (V1), COCO, w/wo DropIT*[7]: batch size 16, SGD optimizer (momentum 0.9), learning rate 0.02, weight decay 10^{-4}, Multistep LR schedule (16,22 epochs), 26 epochs, without AMP training;

• *Mask R-CNN, finetuning from* `torchvision` *ResNet-50 ImageNet-1k weights (V1), COCO, w/wo DropIT*[8]: batch size 16, SGD optimizer (momentum 0.9), learning rate 0.02, weight decay 10^{-4}, Multistep LR schedule (16,22 epochs), 26 epochs, without AMP training.

[4] https://github.com/facebookresearch/deit/blob/main/README_deit.md

[5] https://github.com/facebookresearch/deit/issues/45

[6] https://github.com/facebookresearch/deit/issues/45

[7] https://github.com/pytorch/vision/tree/main/references/detection#faster-r-cnn-resnet-50-fpn

[8] https://github.com/pytorch/vision/tree/main/references/detection#mask-r-cnn