Differentially Private Optimization on Large Model at Small Cost

Zhiqi Bu [1] **Yu-Xiang Wang** [2] **Sheng Zha** [1] **George Karypis** [1]

Abstract

Differentially private (DP) optimization is the standard paradigm to learn large neural networks that are accurate and privacy-preserving. The computational cost for DP deep learning, however, is notoriously heavy due to the per-sample gradient clipping. Existing DP implementations are $2 \sim 1000\times$ more costly in time and space complexity than the standard (non-private) training. In this work, we develop a novel Book-Keeping (BK) technique that implements existing DP optimizers (thus achieving the same accuracy), with a substantial improvement on the computational cost. Specifically, BK enables DP training on large models and high dimensional data to be roughly as fast and memory-saving as the standard training, whereas previous DP algorithms can be inefficient or incapable of training due to memory error. The computational advantage of BK is supported by the complexity analysis as well as extensive experiments on vision and language tasks. Our implementation achieves state-of-the-art (SOTA) accuracy with very small extra cost: on GPT2 and at almost the same memory cost ($< 1\%$ overhead), BK has $1.03\times$ the time complexity of the standard training ($0.83\times$ training speed in practice), and $0.61\times$ the time complexity of the most efficient DP implementation ($1.36\times$ training speed in practice). We open-source the codebase for the BK algorithm at `https://github.com/awslabs/fast-differential-privacy`.

1 Introduction

Deep learning with differential privacy (DP; (Dwork et al., 2006)) has shown strong performance while guaranteeing rigorous protection against privacy risks, especially on large models that tend to memorize and leak the training data (Carlini et al., 2021; Haim et al., 2022; Shokri et al., 2017). For example, recent advances have shed light on the success of DP GPT2 (Li et al., 2021; Bu et al., 2022b; Yu et al., 2021), which achieves 64.6 BLEU score[1] at strong privacy guarantee ($\epsilon = 3$), on the text generation task using E2E restaurant review dataset. This is only marginally below the standard non-private GPT2 (BLEU score 66.8). Similarly, on computer vision tasks ($\epsilon = 2$), DP vision transformers and ResNets have obtained $97.1\%/86.2\%$ accuracy on CIFAR10/100 by (Bu et al., 2022a) and over 81% accuracy on ImageNet by (De et al., 2022; Mehta et al., 2022).

However, DP training of large neural networks is well-known to be computationally burdensome in comparison to the standard training, in terms of both the training time and the memory cost. For instance, training a small recurrent neural network (0.598M parameters) experiences a $1000\times$ slowdown using DP optimizers in Tensorflow-Privacy (TF-Privacy) library in (Bu et al., 2021a), and training a small convolutional neural network (CNN, 0.605M parameters) on CIFAR10 has a $24\times$ slowdown with Tensorflow 2 and the XLA compiler (Subramani et al., 2021). Even with SOTA efficient implementations, large models such as RoBERTa (Liu et al., 2019), GPT2 (Radford et al., 2019), ResNet (He et al., 2016), VGG (Simonyan & Zisserman, 2014), ViT (Dosovitskiy et al., 2020) and its variants, experience about $2 \sim 3\times$ slowdown in Pytorch (Li et al., 2021; Bu et al., 2022a) and $2 \sim 9\times$ slowdown in JAX (Kurakin et al., 2022; De et al., 2022), with possibly $4 \sim 20\times$ memory overhead (Bu et al., 2022a; Li et al., 2021; Subramani et al., 2021) if not running out of memory.

The efficiency bottleneck in DP deep learning lies in the per-sample gradient clipping, which restricts the magnitude of each per-sample gradient in the mini-batch. Applying the clipping jointly with the Gaussian noise addition, one can privately release the gradient to arbitrary optimizers like SGD and Adam, and thus guarantee the privacy of the

[1]Amazon Web Services [2]University of California, Santa Barbara. Correspondence to: Zhiqi Bu <zhiqibu@amazon.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

[1]BLEU (BiLingual Evaluation Understudy) is a metric (0-100) for automatically evaluating translated text. BLEU > 60 is considered as "very high quality, adequate, and fluent translations, often better than human".

Dataset	SOTA setting	Model	Time /Epoch	Relative Speed (same memory contraint) to GhostClip	to Opacus	to non-DP
QQP	(Li et al., 2021)	RoBERTa-large (355M)	70'04"	1.36×	1.96×	0.77×(0.89×)
E2E	(Li et al., 2021)	GPT2-large (774M)	10'01"	1.36×	4.41×	0.83×(0.97×)
CIFAR	(Bu et al., 2022a)	BEiT-large (304M)	6'35"	1.33×	38.3×	0.76×(0.92×)

Table 1. Efficiency of BK algorithm on DP tasks using one A100 GPU (same accuracy). Note the speed is measured in wall-time (hardware speed) and in complexity (theoretical speed). More models and tasks can be found in Table 9.

training as described in Section 1.3:

$$\text{private gradient: } \hat{\mathbf{G}} := \sum_i \boldsymbol{g}_i \cdot C(\|\boldsymbol{g}_i\|_2) + \sigma_{\text{DP}} \cdot \mathcal{N}(0, \mathbf{I}),$$
$$\text{private optimizer (e.g. SGD): } \quad \mathbf{W}_{t+1} = \mathbf{W}_t - \eta \hat{\mathbf{G}}. \quad (1)$$

Here $\mathbf{W}$ is the model parameters, $\mathcal{L}_i$ is the per-sample loss, $\boldsymbol{g}_i = \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}$ is the per-sample gradient, η is the learning rate, σ_{DP} is the noise magnitude that defines the privacy loss, and $C(\|\boldsymbol{g}_i\|)$ or simply C_i is the per-sample clipping factor. For example, in (Abadi et al., 2016), $C_i = \min\{R/\|\boldsymbol{g}_i\|, 1\}$ for some clipping threshold R; in (Bu et al., 2021b), $C_i = \mathbb{I}(\|\boldsymbol{g}_i\| \leq R)$; in (Bu et al., 2022b), $C_i = 1/(\|\boldsymbol{g}_i\| + 0.01)$ or $1/\|\boldsymbol{g}_i\|$ as the gradient normalization.

At high level, the DP training is a system effort consisting of multiple parts:

I. optimizer: DP-SGD, DP-Adam, DP-LAMB;

II. parameter efficiency: last layer (linear probing), LoRA, Adapter, BiTFiT;

III. implementation: Opacus, GhostClip, Book-Keeping;

IV. platform: Pytorch, JAX, TensorFlow (TF).

Previous works have tackled the efficiency bottleneck with various approaches. One approach (part II) focuses on the parameter efficiency by partially training a neural network, in contrast to fully fine-tuning all model parameters, e.g. only the last output layer (Tramer & Boneh, 2020), the adapter layers (Houlsby et al., 2019; Mahabadi et al., 2021), or the Low-Rank Adaptation (LoRA) (Hu et al., 2021; Yu et al., 2021). For example, (Mehta et al., 2022) accelerate the DP training on ImageNet (Deng et al., 2009) up to $30\times$ by only training the last layer of ResNet152. Noticeably, parameter efficient fine-tuning does not improve on the efficiency in terms of complexity per parameter, rather than reducing the number of parameters. Furthermore, this approach oftentimes leads to some accuracy degradation compared to DP full fine-tuning (Bu et al., 2020; Mehta et al., 2022; Li et al., 2021; Yu et al., 2021).

An orthogonal approach, including this work, focuses on the computation efficiency (part III), i.e. reducing the time and space complexity through efficient implementations, without modifying the DP optimizers (part I) and thus not affecting their performance. We will elaborate on existing methods in Section 1.2. Additionally, these methods can be compiled on different platforms (part IV) such as Tensorflow 2(XLA), JAX and Pytorch (Li et al., 2021; Subramani et al., 2021; De et al., 2022; Kurakin et al., 2022), where remarkable speed difference has been observed in some cases, even with the same implementation. For example, (Subramani et al., 2021) implemented DP-SGD using JAX and claimed its efficiency advantage over the same algorithm using Tensorflow or Pytorch.

1.1 Contributions

1. **[Algorithm]** We propose the book-keeping (BK) algorithm that makes existing DP optimizers fast and memory efficient, especially comparable to non-private optimizers. We demonstrate BK via the computation graph in Figure 1. The highlight is that BK *only uses one back-propagation* and *never instantiates per-sample gradients* $\{\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\}_{i=1}^B$.

2. **[Analysis]** We analyze the complexity to show that *BK has almost the same time and space complexity as non-DP training*, especially when the feature dimension is small (see Table 5).

3. **[Extension]** We strengthen BK using a layerwise decision to mix with Opacus (see Section 3.2), which proves to be efficient when the feature dimension is large (and difficult for GhostClip). We also extend BK to the parameter efficient fine-tuning such as DP LoRA and Adapter.

4. **[Codebase]** We develop a Pytorch (Paszke et al., 2019) codebase for our BK algorithm, leveraging the auto-differentiation technique on the computation graph and a new trick in Appendix D.2. We highlight that our codebase can automatically switch the standard training of *any model* to its DP version, by adding a single piece of codes.

5. **[Experiments]** We demonstrate the amazing efficiency of BK on training large models, saving the memory up to $10\times$ and boosting the speed by $30\% \sim 5\times$ than previous DP implementations.

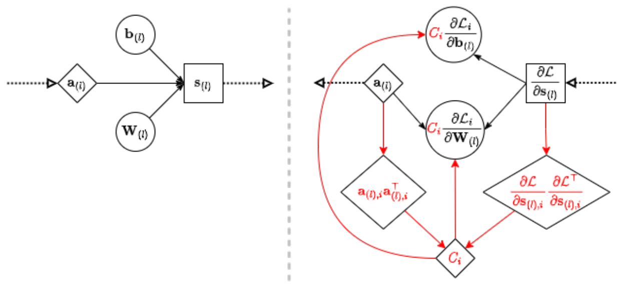

Figure 1. Forward pass and back-propagation of the l-th linear layer (standard training is in black; DP training by our book-keeping algorithm is added in red). Here $\boldsymbol{a}_{(l)}$ is the activation tensor, $\boldsymbol{s}_{(l)}$ is the layer output, $\mathbf{W}_{(l)}, \mathbf{b}_{(l)}$ are weight and bias, $\mathcal{L}_i, \mathcal{L}$ are the per-sample loss and the summed loss. The dotted arrow is the inter-layer operation such as pooling or normalization.

1.2 Related works

Previous arts have developed different implementations of the same DP optimizer in Equation (1). Among these implementations, the tradeoff between the time and space complexity has been constantly maneuvered. TF-Privacy (Tensorflow) back-propagates a vectorized loss $[\mathcal{L}_1, \cdots, \mathcal{L}_B]$ to compute the per-sample gradients, each from one back-propagation, which is memory-efficient but slow. Opacus (Yousefpour et al., 2021) and (Rochette et al., 2019) accelerate the training significantly using the outer product trick in (Goodfellow et al., 2014), though incurring heavy memory burden so as to store the per-sample gradients. This memory burden is partially alleviated in FastGradClip (Lee & Kifer, 2020) by sharing the space complexity in two rounds of back-propagation, hence almost doubling the time complexity. In ghost clipping (Goodfellow, 2015; Li et al., 2021; Bu et al., 2022a), the per-sample gradients can be clipped without being instantiated, thus both time and space complexity can be further improved if the feature dimension is small. We refer interested readers to Figure 3 and Appendix C for algorithmic details of these implementations.

We now compare BK to different implementations in Table 2 and Figure 2. In what follows, B is the batch size[2], $T_{(l)}$ is the feature dimension[3], $d_{(l)}, p_{(l)}$ are the input or output dimension of a layer.

1.3 Preliminaries

We work with the (ϵ, δ)-DP by (Dwork et al., 2006), defined in Appendix A, which makes it difficult for any privacy attacker to distinguish or detect an arbitrary training sample, even with full access to the model. In deep learning, DP is achieved by training on the private gradient in Equation (1) with any optimizer such as SGD, Adam, FedAvg, etc. Essentially, the private gradient is the addition of Gaussian noise to the sum of clipped per-sample gradients, which guarantees the DP protection through the privacy accounting theorems (Abadi et al., 2016; Mironov, 2017; Dong et al., 2019; Zhu et al., 2021; Gopi et al., 2021; Koskela et al., 2020).

2 Book-keeping: Efficient DP training in low dimension

The main computational bottleneck of DP training comes from the per-sample gradient clipping, or from the computation of per-sample gradient norms, to be exact. One widely used approach in Opacus, TF-privacy, and FastGradClip, is to instantiate the per-sample gradients and then deriving their norms. Straight-forward implementation of this approach on a mini-batch of per-sample losses requires B rounds of back-propagation (unacceptable slowdown) or $B\times$ gradient storage (unacceptable memory burden; see Opacus in Figure 2). Consequently, these implementations are not suitable for large model training. For instance, (Li et al., 2021) shows that, when training GPT2-large (774M

[2] In this work, we report the physical batch size, which affects the efficiency but not the accuracy; the accuracy is only affected by the logical batch size, which can be implemented through the gradient accumulation of physical batch size.

[3] For non-sequential data, $T = 1$; for texts, T is the sequence length, which is layer-independent; for images (or videos), $T_{(l)}$ is the height×width(×time) of hidden feature representation, which is layer-dependent.

	Non-DP	TF-privacy	Opacus	FastGradClip	GhostClip	BK (ours)
Instantiating per-sample grad	✗	✓	✓	✓	✗	✗
Storing every layer's grad	✗	✗	✓	✗	✗	✗
Instantiating non-DP grad	✓	✓	✓	✗	✓	✗
Number of back-propagation	1	B	1	2	2	1
Time Complexity of Clipping	$6BTpd$	$6BTpd$	$8BTpd$	$8BTpd$	$10BTpd + O(BT^2)$	$\approx 6BTpd$
Memory Overhead to non-DP	0	0	Bpd	Bpd	$2BT^2$	$\min\{2BT^2, Bpd\}$
Scalable to large model	✓	✗	✗	✗	✓	✓
Scalable to high-dim input	✓	✗	✓	✓	✗	✓

Table 2. Summary of different DP implementations on a linear/convolution layer $\mathbb{R}^{B\times T_{(l)}\times d_{(l)}} \to \mathbb{R}^{B\times T_{(l)}\times p_{(l)}}$. The main bottleneck is marked in red.

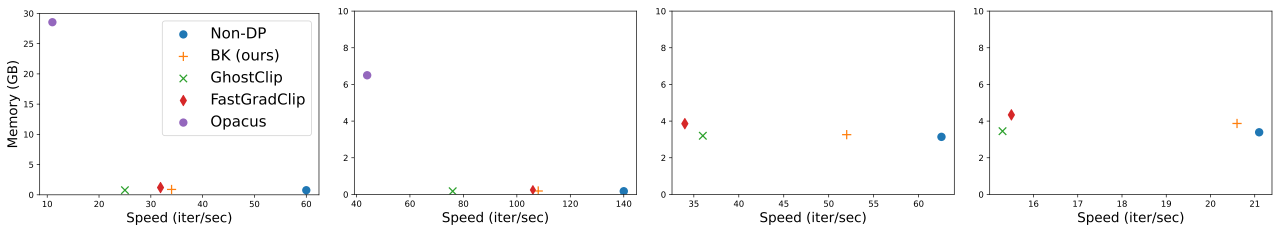

Figure 2. Speed and memory on MLP and CIFAR100 (images are flattened into vectors). Left to right: deep network (50 layers, width 1000, 50M parameters, batch size 128), shallow network (10 layers, width 1000, 10M parameters, batch size 128), and wide network (10 layers, width 5000, 250M parameters, batch size 128 or 1024; Opacus is OOM). See more ablation study in Appendix F.

parameters), Opacus (Yousefpour et al., 2021) and JAX (Subramani et al., 2021) cannot fit even one single sample into a 24GB GPU.

An alternative approach, termed as the ghost clipping (GhostClip), directly computes the per-sample gradient norms without computing the gradients themselves. This is made possible, unfortunately, through two rounds of back-propagation. During the first back-propagation, one uses the regular loss $\sum_i \mathcal{L}_i$ and extracts the activation tensor and the output gradient $(\boldsymbol{a}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}})$. One can use an algebraic trick in Equation (2) to compute the per-sample gradient norms $\{\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|\}_i$ and the clipping factors $\{C_i\}_i$ in Equation (1). During the second back-propagation, one uses the reweighted loss $\sum_i C_i \mathcal{L}_i$ whose gradient is directly the weighted gradient $\sum_i C_i \boldsymbol{g}_i$, which constitutes the private gradient we need. Note that this double back-propagation roughly doubles the training time (or to be more precise, $10/6 \approx 1.667\times$ when T is small; but this approach loses its advantage when T is large), as shown in Table 2).

To make the DP training as efficient as the standard training, we propose the book-keeping technique (BK) that $\langle 1 \rangle$ only requires a single round of back-propagation, like Opacus and the standard training; $\langle 2 \rangle$ does not instantiate the per-sample gradients, like GhostClip.

2.1 Book-keeping algorithms

BK algorithms in their base forms are built on GhostClip and especially the *ghost norm* trick, so as to avoid instantiating the memory costly per-sample gradients: as can be seen in Algorithm 1 and Figure 3, $\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}} = \boldsymbol{a}_i^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}$ is not computed throughout the training. In comparison to GhostClip, our significant improvement is solely on the speed (see Table 2) through two novel tricks: the *book-keeping* and the *ghost differentiation*. The entire BK algorithm is built on the understanding of computation graph in Appendix A. Note that these tricks also offer improved efficiency for existing implementations, to be presented in Section 2.4. We now elaborate on these tricks.

$$\text{BK (base)} = \underbrace{\text{ghost norm}}_{\text{from GhostClip}} + \underbrace{\text{book-keeping}}_{\text{ours}} + \underbrace{\text{ghost differentiation}}_{\text{ours}}$$

Algorithm 1 Differentially private deep learning with BK

Parameters: l-th layer weights $\mathbf{W}_{(l)}$, number of layers L, noise level σ.

for layer $l \in 1, 2, \cdots, L$ **do**
  Get activation tensor $\{\boldsymbol{a}_{(l),i}\}$ by forward hook
for layer $l \in L, L-1, \cdots, 1$ **do**
  Get output gradient $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$ by backward hook
  Compute per-example gradient norm $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$ by ghost norm trick in Equation (2)
Aggregate gradient norm across layers: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F^2 = \sum_l \|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
Compute clipping factor: $C_i = C(\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F; R)$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \text{diag}(C_1, C_2, \cdots) \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
Add Gaussian noise $\hat{\mathbf{G}} = \mathbf{G} + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
Apply SGD/Adam/LAMB with the private gradient $\hat{\mathbf{G}}$

Ghost norm trick The ghost norm trick (Goodfellow, 2015) computes the gradient norm without the gradient: while the gradient is instantiated by the multiplication in Equation (2), the gradient norm can be computed without $\boldsymbol{a}_i$ meeting $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}$. This trick is applicable to generalized linear layers including the linear, the embedding (Li et al., 2021), and the convolution layers (Bu et al., 2022a). We emphasize that these generalized linear layers represent 99.9% of the trainable parameters in modern neural networks.

We demonstrate this trick using a simple linear layer $\boldsymbol{s}_i = \boldsymbol{a}_i \mathbf{W}$, where $\mathbf{W} \in \mathbb{R}^{d \times p}$ is the weight matrix, $\boldsymbol{a} \in \mathbb{R}^{B \times T \times d}$ is the mini-batch input of this layer (a.k.a. the activation tensor) and $\boldsymbol{s} \in \mathbb{R}^{B \times T \times p}$ is the output. Given that the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$ is readily available in the back-propagation, for DP and standard training, one can directly derive the per-sample gradient norm

$$\left\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\right\|_{\text{Frobenius}}^2 = \text{vec}\Big(\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i} \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}^\top\Big) \cdot \text{vec}\left(\boldsymbol{a}_i \boldsymbol{a}_i^\top\right) \quad (2)$$

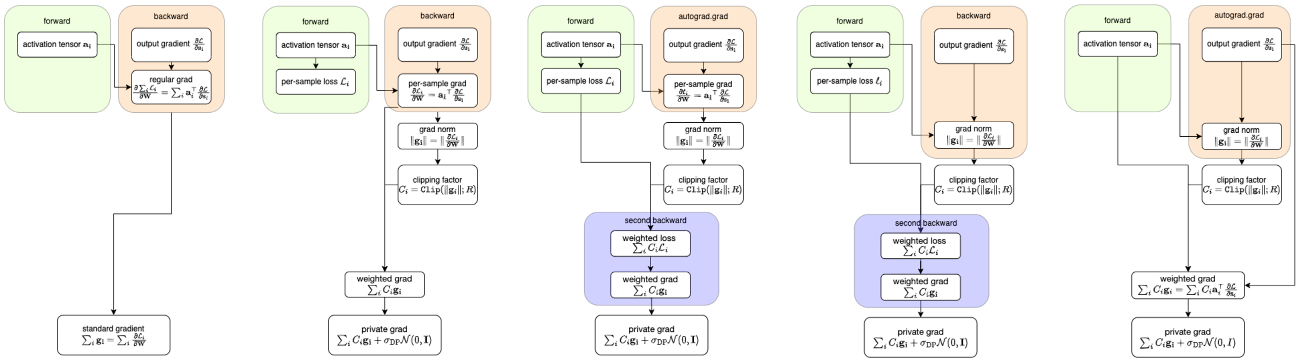

Figure 3. Standard (non-DP), Opacus, FastGradClip, GhostClip, and BK implementations, from left to right. Notice that BK directly computes clipped gradient like Opacus, computes the ghost norm like GhostClip, and uses auto-differentiation like FastGradClip.

without actually computing $\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}} = \boldsymbol{a}_i^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}$. Here 'vec' means flattening the $T \times T$ matrix to a vector. This trick is particularly efficient when T is small, reducing the space complexity from $O(Bpd)$ to $O(BT^2)$ by Table 3.

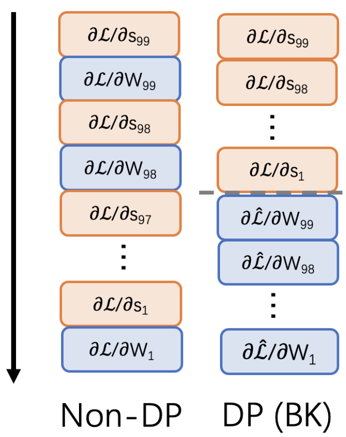

Figure 4. Backward propagation of BK algorithm. Here $\mathcal{L} := \sum_i \mathcal{L}_i, \hat{\mathcal{L}} := \sum_i C_i \mathcal{L}_i$.

Book-keeping trick This trick improves the time complexity by removing the second back-propagation from GhostClip. Our idea is to book-keep and re-use the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$, which is deleted after the first back-propagation of GhostClip and must be re-computed during the second back-propagation. The difference between GhostClip and BK is clearly illustrated via a line-by-line comparison in Appendix C.1. In fact, denoting the total number of model parameters as $M = \sum_l p_{(l)} d_{(l)}$, our trick reduces the time complexity from $10BTM + O(BT^2)$ by GhostClip to $8BTM + O(BT^2)$ according to Table 3. In contrast to Opacus, which book-keeps the per-sample gradients $\boldsymbol{g}_i^{(l)}$ using $O(BM) = O(B \sum_l p_{(l)} d_{(l)})$ memory, we instead book-keep the output gradient with substantially cheaper $O(BT \sum_l p_{(l)})$ memory when the feature dimension T is small.

Ghost differentiation trick This trick improves the time complexity on the first back-propagation in GhostClip, further reducing from $8BTM + O(BT^2)$ to $6BTM + O(BT^2)$ in Table 2. Our idea is to only compute the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$ but not the (non-private) parameter gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$. That is, we break the $4BTM$ time complexity of the full back-propagation into two sub-processes, each of $2BTM$ complexity, and remove the unnecessary one.

To be more specific, during the back-propagation of Opacus and GhostClip, the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$ and then the parameter gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \boldsymbol{a}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$ are computed. However, we can stop after we obtain $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$: we only need the output gradient to compute the clipped parameter gradient $\frac{\partial \sum_i C_i \mathcal{L}_i}{\partial \mathbf{W}}$ in Line 9 of Algorithm 1. Therefore, the ghost differentiation trick sets all parameters to not require gradients. See the technical details in Appendix D.2, including the *origin parameter trick* that propagates on a computation graph even when no parameters require gradients.

2.2 Complexity of DP implementations: a modular analysis

In this section, we analyze the complexity of DP implementations from their opearation modules. We summarize the time and space complexity in Table 3 and give the derivation in Appendix B. We will refer to these modules by indices, e.g. 2a for the computation of output gradient.

Now we are ready to decompose each implementation, following the flowcharts in Figure 3. Consequently, we can easily write down the complexity of different implementations in Table 2. Such a modular analysis displays the clear effects of the tricks in BK algorithm: the ghost norm trick removes the memory costly 4 from Opacus and FastGradClip; the book-keeping trick removes the 2b in the second back-propagation of FastGradClip and GhostClip; the ghost differentiation trick removes the 2b in the first back-propagation of Opacus and GhostClip.

- Standard (non-DP)= 1 + 2a + 2b
- Opacus= 1 + 2a + 2b + 4 + 5
- FastGradClip= 1 + 2a + 4 + 2a + 2b
- GhostClip= 1 + 2a + 2b + 3 + 2a + 2b
- BK (ours)= 1 + 2a + 3 + 2b

2.3 BK is optimally efficient in low dimension

When the feature dimension T is small, we claim that BK is almost as efficient as the standard non-private training, with a negligible $O(BT^2)$ time and memory overhead by Table 2:

$$\begin{aligned}\textbf{Memory complexity: }&\text{non-DP} \approx \text{BK} \approx \text{GhostClip}\\&< \text{FastGradClip} \ll \text{Opacus}\\\textbf{Time complexity: }&\text{non-DP} \approx \text{BK} < \text{FastGradClip}\\&\approx \text{Opacus} < \text{GhostClip}\end{aligned}$$

Now, we discuss the cases where the data has low dimension and thus T is small. Generally speaking, the feature dimension $T_{(l)}$ depends on both the data and the model.

For non-sequential input and 1D audio data, $T = 1$. For sequential data such as texts (T being sentence length) or time series (T being time duration), $T_{(l)}$ is fixed across layers. In this case, BK is efficient on short-sequence datasets including GLUE (Wang et al., 2019) (e.g. SST2/QNLI/MNLI/QQP) and natural language generation datasets (e.g. E2E/DART), since $T^2 \ll p_{(l)}d_{(l)}$. For instance, (Yu et al., 2021; Li et al., 2021; Bu et al., 2022b) applied GPT2 on E2E dataset, which has a sequence length $T \approx 100$ and the number of parameters $p_{(l)}d_{(l)}$ per layer is $2-4$M; (Yu et al., 2021; Li et al., 2021) applied RoBERTa-large on GLUE datasets, which has a sequence length $T = 256$ and the number of parameters per layer is $1-4$M. As illustrated in Figure 5 and Table 1 (extended in Table 9), BK improves the throughput of existing implementations by $25-388\%$ on multiple language tasks in (Li et al., 2021; Bu et al., 2022b), with minor memory overhead compared to GhostClip and non-private training.

However, on the convolution layers with image data, $T_{(l)}$ is the product of hidden feature sizes (c.f. Section 3 in (Bu et al., 2022a)), thus $T_{(l)}$ depends on the original image size and network architecture. For example, larger kernel size/dilation/stride in convolution layer reduces $T_{(l)}$, while larger images have larger $T_{(l)}$ at each layer. Therefore, BK (and GhostClip) may suffer on when training ResNet on ImageNet (224×224), as we show in Figure 6 (see also Table 7 in (Bu et al., 2022a)), although training the same network efficiently on CIFAR10/100 (32×32).

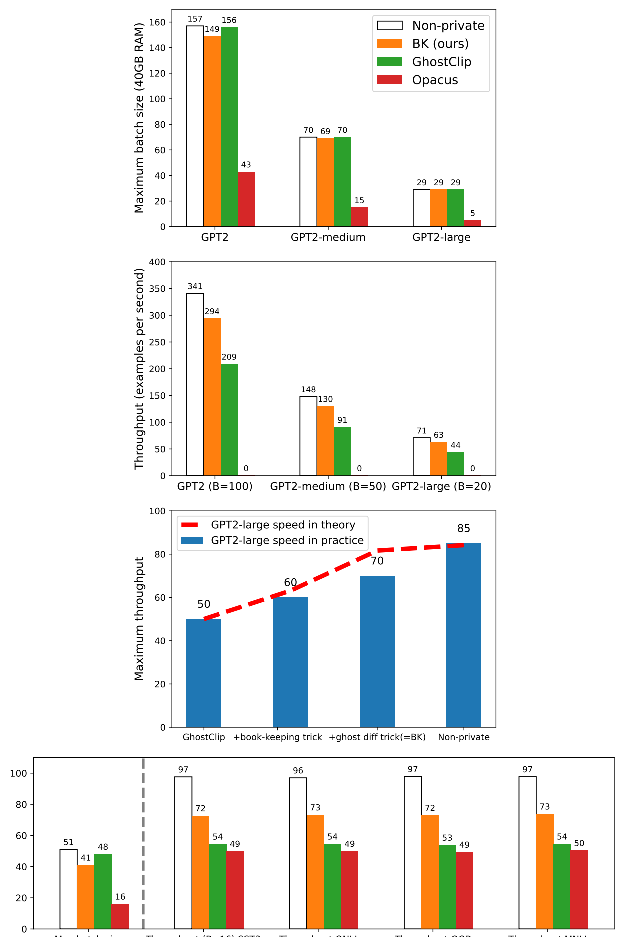

Figure 5. Memory and speed of different DP implementations. Upper: GPT2 on E2E dataset (fixing B, DP speed is $0.86 \sim 0.89\times$ of non-DP). Lower: RoBERTa-large on GLUE datasets. Note here the hybrid implementations are equivalent to the base ones, because of the short sequence length.

Module	1 Forward pass	2 Back-propagation (a)output gradient	(b)parameter gradient	3 Ghost norm	4 Per-sample grad instantiation	5 Weighted sum of per-sample grad
Time complexity	$2BTpd$	$2BTpd$	$2BTpd$	$2BT^2(p+d)$	$2BTpd$	$2Bpd$
Space complexity	$pd+BTd$	$BT(p+d)$	pd	$2BT^2$	Bpd	0

Table 3. Time and space complexity of modules in DP training for one generalized linear layer.

2.4 Applying our tricks to existing implementations

Our tricks in Section 2.1 can also improve other existing implementations, reducing the time complexity of GhostClip from $10BTpd + 2BT^2(p+d)$ to $6BTpd + 2BT^2(p+d)$, that of Opacus and FastGradClip from $8BTpd$ to $6BTpd$. We highlight that these improved implementations are leveraged to design hybrid implementation in Section 3.2. In addition to DP full fine-tuning, BK is demonstrated in Appendix E.2 to also apply to the parameter efficient fine-tuning like Adapters (Houlsby et al., 2019) and LoRA (Hu et al., 2021).

$$\begin{aligned}
\text{GhostClip} &= 1 + 2a + 2b + 3 + 2a + 2b \\
&\xrightarrow[\text{book-keeping}]{\text{ghost differentiation}} 1 + 2a + 3 + 2b \ \text{(BK)} \\
\text{Opacus} &= 1 + 2a + 2b + 4 + 5 \\
&\xrightarrow{\text{ghost differentiation}} 1 + 2a + 4 + 5 \\
\text{FastGradClip} &= 1 + 2a + 4 + 2a + 2b \\
&\xrightarrow{\text{book-keeping}} 1 + 2a + 4 + 2b
\end{aligned}$$

3 Hybrid Book-keeping: Efficient DP training in high dimension

In previous section, we have analyzed DP implementations in the small T regime, where the ghost norm-based GhostClip and BK are efficient. Nevertheless, in the large T and large model regime, none of the base implementations may be efficient (see Figure 6) and we turn to hybrid methods.

3.1 Large T necessitates non-ghost norm method

A closer look at the space complexity in Table 3 shows that, the ghost norm trick is favored over the per-sample gradient instantiation if and only if $2T_{(l)}^2 < p_{(l)}d_{(l)}$, where $p_{(l)}d_{(l)}$ is the number of parameters at one layer. When this criterion is violated for large T, GhostClip/BK (base) can significantly under-perform Opacus/FastGradClip, as shown in Figure 6, Figure 7 and Table 10.

Similar to Section 2.3, we discuss two cases where T is large. For paragraph or document-level language tasks like WikiHop (Welbl et al., 2018) and TriviaQA (Joshi et al., 2017), T can range from $2000 \sim 20000$ to train large language models, which makes $2T^2 = 8 \sim 800$M. For example, LLAMA (Touvron et al., 2023) is trained with token length $4096 \leq T \leq 8192$ and GPT-3 (Brown et al., 2020) is trained with token length $T = 2048$.

For image tasks, particularly on CNN, $T_{(l)}$ varies at each layer with large values on top layers, as the features are less compressed by convolution and pooling. Taking ImageNet and the first convolution layer of VGG11 as an example (see Table 3 of (Bu et al., 2022a)), $2T_{(1)}^2 = 5 \times 10^9 \gg p_{(1)}d_{(1)} = 1.7 \times 10^3$. Consequently, ghost norm-based implementations (i.e. GhostClip and BK) costs more than 40GB memory on ResNet18, under $B = 32$, while Opacus only costs 2.5GB. This curse of dimension grows from a difficult issue on ImageNet to an impossible challenge on videos or high-resolution images, e.g. GhostClip cannot train ResNet18 with even one single CelebA-HQ image (1024×1024) using a 40GB GPU.

In short, the ghost norm trick is inefficient for large T and the per-sample gradient instantiation is inefficient for large model. Hence, we must hybridize the base implementations.

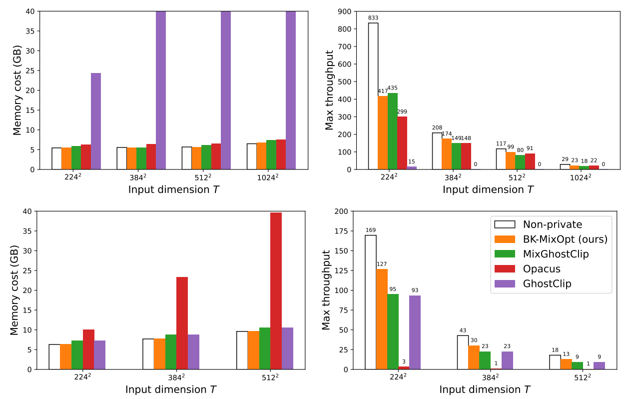

Figure 6. Memory and speed by different implementations on 50000 images. Top is VGG11 (133M; (Simonyan & Zisserman, 2014)). Bottom is BEiT-large (304M; (Bao et al., 2021)). Memory cost is measured with physical batch size 1. Throughput is measured with the maximum physical batch size on 40GB memory.

3.2 Hybrid implementations via layerwise decision

We adopt the same layerwise decision as (Bu et al., 2022a), known as the mixed ghost norm technique: we use the ghost norm trick on a layer if $2T_{(l)}^2 < p_{(l)}d_{(l)}$, and instantiate per-sample gradients otherwise. Therefore, the space complexity of computing the per-sample gradient norm reduces to $\min\{2T_{(l)}^2, p_{(l)}d_{(l)}\}$, which is significantly cheaper than either the ghost norm or the per-sample gradient instantiation in high dimension, as depicted in Table 4 and Figure 7. Consequently, over all layers, the space complexity is lower than both constituting methods, e.g. saving more than $10\times$ memory for the per-sample gradient clipping on ResNet18 (see more models in Table 10).

In contrast to the mixed ghost clipping (MixGhostClip) in (Bu et al., 2022a), which hybridizes FastGradClip and GhostClip, we boost the efficiency by hybridizing our BK with the improved FastGradClip/Opacus in Section 2.4. We propose BK-MixOpt (and BK-MixGhostClip as an intermediate product only for comparison) and use MixGhostClip as a reference point,

	Output size	Space complexity					
		18-layer		34-layer		50-layer	
	$H_{\text{out}} \times W_{\text{out}}$	Ghost norm $2T_{(l)}^2 = 2H_{\text{out}}^2 W_{\text{out}}^2$	Per-sample grad instantiation $p_{(l)} d_{(l)} =$ # params	Ghost norm $2T_{(l)}^2$	Per-sample grad instantiation $p_{(l)} d_{(l)}$	Ghost norm $2T_{(l)}^2$	Per-sample grad instantiation $p_{(l)} d_{(l)}$
conv1	112×112	3.1×10^8	$\mathbf{9.4 \times 10^3}$	3.1×10^8	$\mathbf{9.4 \times 10^3}$	3.1×10^8	$\mathbf{9.4 \times 10^3}$
conv2_x	56×56	$[2.0 \times 10^7] \times 4$	$\mathbf{[3.7 \times 10^4] \times 4}$	$[2.0 \times 10^7] \times 6$	$\mathbf{[3.7 \times 10^4] \times 6}$	$[2.0 \times 10^7] \times 9$	$\mathbf{[4.1 \times 10^3] \times 1}$ $\mathbf{[3.7 \times 10^4] \times 3}$ $\mathbf{[1.6 \times 10^4] \times 5}$
conv3_x	28×28	$[1.2 \times 10^6] \times 4$	$\mathbf{[7.4 \times 10^4] \times 1}$ $\mathbf{[1.5 \times 10^5] \times 3}$	$[1.2 \times 10^6] \times 8$	$\mathbf{[7.4 \times 10^4] \times 1}$ $\mathbf{[1.5 \times 10^5] \times 7}$	$[2.0 \times 10^7] \times 1$ $[1.2 \times 10^6] \times 11$	$\mathbf{[3.3 \times 10^4] \times 1}$ $\mathbf{[6.6 \times 10^4] \times 7}$ $\mathbf{[1.5 \times 10^5] \times 4}$
conv4_x	14×14	$\mathbf{[7.7 \times 10^4] \times 4}$	$[2.9 \times 10^5] \times 1$ $[5.9 \times 10^5] \times 3$	$\mathbf{[7.7 \times 10^4] \times 12}$	$[2.6 \times 10^5] \times 1$ $[5.9 \times 10^5] \times 11$	$[1.2 \times 10^6] \times 1$ $\mathbf{[7.7 \times 10^4] \times 17}$	$\mathbf{[1.3 \times 10^5] \times 1}$ $[2.6 \times 10^5] \times 11$ $[5.9 \times 10^5] \times 6$
conv5_x	7×7	$\mathbf{[4.8 \times 10^3] \times 4}$	$[1.2 \times 10^6] \times 1$ $[2.4 \times 10^6] \times 3$	$\mathbf{[4.8 \times 10^3] \times 6}$	$[1.2 \times 10^6] \times 1$ $[2.4 \times 10^6] \times 5$	$\mathbf{[4.8 \times 10^3] \times 9}$	$[5.2 \times 10^5] \times 1$ $[1.0 \times 10^6] \times 5$ $[2.4 \times 10^6] \times 3$
linear	1×1	$\mathbf{2}$	5.1×10^5	$\mathbf{2}$	5.1×10^5	$\mathbf{2}$	2.0×10^6
Total complexity		399M	11.5M	444M	21.6M	528M	22.7M
Complexity by mixed ghost norm		1.0M		2.3M		2.8M	

Table 4. Space complexity of the per-sample gradient clipping (not the entire DP algorithm) for $B = 1$ on ImageNet 224×224. Layerwise decision of hybrid BK algorithms is highlighted in **bold**.

Method	Type	Modification to previous variant	Time complexity	Space complexity
Non-DP			$6BTpd$	$pd + 3BTd + BTp$
Opacus	base	Instantiate per-sample gradient	$8BTpd$	$+Bpd$
FastGradClip	base	Not store per-sample gradient using a second back-propagation	$8BTpd$	$+Bpd$
GhostClip	base	Not instantiate per-sample gradient using ghost norm trick	$10BTpd + 2BT^2(p+d)$	$+2BT^2$
BK (ours)	base	Simplify the two back-propagations	$6BTpd + 2BT^2(p+d)$	$+2BT^2$
MixGhostClip	hybrid	Mix ways to compute grad norm	$8BTpd + \langle 2BTpd, 2BT^2(p+d)\rangle$	$+\min\{2BT^2, Bpd\}$
BK-MixGhostClip	hybrid	Mix ways to compute grad norm	$6BTpd + \langle 2BTpd, 2BT^2(p+d)\rangle$	$+\min\{2BT^2, Bpd\}$
BK-MixOpt	hybrid	Mix ways to compute weighted grad	$6BTpd + \langle 0, 2BT^2(p+d)\rangle$	$+\min\{2BT^2, Bpd\}$

Table 5. Complexity of DP implementations on one layer. Here $\langle\rangle$ means between two values. The exact time complexity of BK-MixOpt is $6BTpd + 2BT^2(p+d) \cdot \mathbb{I}\{2T^2 < pd\} \approx 6BTpd$. The space complexity of DP algorithms is in addition to that of non-DP one.

- MixGhostClip $= 1 + 2a + 2b + \min\{3, 4\} + 2a + 2b \approx \min\{$GhostClip, FastGradClip$\}$,
- BK-MixGhostClip $= 1 + 2a + \min\{3, 4\} + 2b = \min\{$BK, improved FastGradClip in Section 2.4$\}$,
- BK-MixOpt $= 1 + 2a + \min\left\{3 + 2b, 4 + 5\right\} = \min\{$BK, improved Opacus in Section 2.4$\}$.

The hybrid BK algorithms are presented in Algorithm 5. We summarize the layerwise complexity in Table 5, from which we derive the overall complexity in Table 8 and observe that BK has almost the same complexity as non-DP training. Note that in low dimension, the mixed ghost norm is equivalent to the ghost norm, hence MixGhostClip/BK-MixOpt is equivalent to GhostClip/BK, respectively.

3.3 Effect of model architecture & feature dimension on hybridization

We dive deeper to understand when the hybridization favors the ghost or non-ghost norm tricks.

From a model architecture viewpoint, transformers such as ViT, RoBERTa, GPT tend to prefer the ghost norm: for moderate-sequence text data and moderate-dimension image data, hybrid BK algorithms are close or equivalent to the base BK algorithm (see right-most plot in Figure 7). However, CNN prefers the per-sample gradient instantiation at top layers, and there exists a depth threshold below which the ghost norm is more efficient. Hence the hybridization is necessary to take advantages of both worlds.

From the feature dimension viewpoint, larger input enlarges

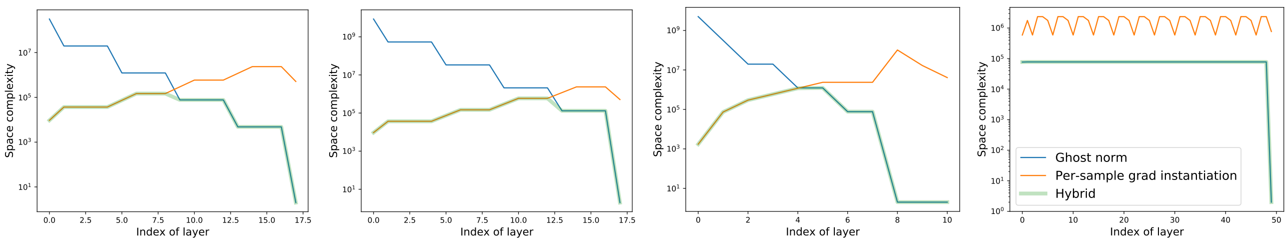

Figure 7. Layerwise space complexity of computing the per-sample gradient norm. Left to right: ResNet18 (224×224), ResNet18 (512×512), VGG11 (224×224), and ViT-base (224×224).

this depth threshold, e.g. from the 9-th layer of ResNet18 to the 17-th layer in Figure 7, when the image size increases from 224×224 to 512×512. We visualize this pattern on various models in Appendix G. In particular, we observe in Table 8 that when T is large, both per-sample gradient instantiation (Opacus) and ghost norm trick (GhostClip) are significantly dominated by our BK algorithms.

4 Instructions to use the codebase

In this section, we demonstrate how to modify a standard training script to its DP variants[4] by one piece of code.

```
from BK import PrivacyEngine
import torch.functional as F

optimizer =
↪  torch.optim.Adam(model.parameters())

privacy_engine = PrivacyEngine(
        model,epochs,
        batch_size,sample_size,
        target_epsilon,target_delta)

privacy_engine.attach(optimizer)

logits = model(data)
loss = F.cross_entropy(logits, labels)
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

We highlight that our codebase automatically modifies the training for any network architecture and any optimizer. Additionally, it is designed to work compatibly with large-scale training techniques, such as the gradient accumulation, the parameter-efficient fine-tuning (e.g. LoRA and BiTFiT (Bu et al., b)), and the parallel distributed learning (e.g. ZeRO (Bu et al., a)).

[4]That is, our codebase can easily adapt to any per-sample gradient clipping function and privacy accouting methods.

5 Discussion

In this work, we propose the Book-Keeping (BK) algorithms to effciently implement DP optimizers using three tricks: ghost norm, book-keeping, and ghost differentiation. Our BK reduces the time and space complexity of DP training to the similar level of the standard training. Specially, we develop hybrid BK to overcome the computational challenge of training large models with high-dimensional data, and we extend BK to parameter efficient fine-tuning such as LoRA and Adapter.

One limitation of this work is that BK (and GhostClip) only applies to the weights, not the biases, and only on the generalized linear layers, i.e. the embedding, the linear, and the convolution layers. However, this is a minor concern because the weights in the generalized linear layers constitute 99.9% of the trainable parameters (see Table 7).

Implementation-wise, our codebase is automatic (allowing any model to be DP optimized) and future-proof (allowing any training setting, including the distributed learning). However, although BK is theoretically as fast as the standard training for small T, we observe some gap between the theoretical complexity and the hardware throughput in practice. This gap is mainly due to the mechanism of Pytorch hooks which can be possibly optimized by customizing the CUDA kernel or using the symbolic programming. We expect this gap to be closed by future research.

References

Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 308–318, 2016.

Bao, H., Dong, L., Piao, S., and Wei, F. Beit: Bert pre-training of image transformers. In *International Conference on Learning Representations*, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners.

Advances in neural information processing systems, 33: 1877–1901, 2020.

Bu, Z., Chiu, J., Liu, R., Wang, Y.-X., Zha, S., and Karypis, G. Zero redundancy distributed learning with differential privacy. In *ICLR 2023 Workshop on Pitfalls of limited data and computation for Trustworthy ML*, a.

Bu, Z., Wang, Y.-X., Zha, S., and Karypis, G. Differentially private bias-term only fine-tuning of foundation models. In *Workshop on Trustworthy and Socially Responsible Machine Learning, NeurIPS 2022*, b.

Bu, Z., Dong, J., Long, Q., and Su, W. J. Deep learning with gaussian differential privacy. *Harvard data science review*, 2020(23), 2020.

Bu, Z., Gopi, S., Kulkarni, J., Lee, Y. T., Shen, H., and Tantipongpipat, U. Fast and memory efficient differentially private-sgd via jl projections. *Advances in Neural Information Processing Systems*, 34, 2021a.

Bu, Z., Wang, H., and Long, Q. On the convergence and calibration of deep learning with differential privacy. *arXiv preprint arXiv:2106.07830*, 2021b.

Bu, Z., Mao, J., and Xu, S. Scalable and efficient training of large convolutional neural networks with differential privacy. *arXiv preprint arXiv:2205.10683*, 2022a.

Bu, Z., Wang, Y.-X., Zha, S., and Karypis, G. Automatic clipping: Differentially private deep learning made easier and stronger. *arXiv preprint arXiv:2206.07136*, 2022b.

Carlini, N., Tramer, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, U., et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2633–2650, 2021.

De, S., Berrada, L., Hayes, J., Smith, S. L., and Balle, B. Unlocking high-accuracy differentially private image classification through scale. *arXiv preprint arXiv:2204.13650*, 2022.

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

Dong, J., Roth, A., and Su, W. J. Gaussian differential privacy. *arXiv preprint arXiv:1905.02383*, 2019.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pp. 265–284. Springer, 2006.

Dwork, C., Roth, A., et al. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

Goodfellow, I. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Gopi, S., Lee, Y. T., and Wutschitz, L. Numerical composition of differential privacy. *Advances in Neural Information Processing Systems*, 34, 2021.

Haim, N., Vardi, G., Yehudai, G., Shamir, O., and Irani, M. Reconstructing training data from trained neural networks. *arXiv preprint arXiv:2206.07758*, 2022.

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

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pp. 2790–2799. PMLR, 2019.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Koskela, A., Jälkö, J., and Honkela, A. Computing tight differential privacy guarantees using fft. In *International Conference on Artificial Intelligence and Statistics*, pp. 2560–2569. PMLR, 2020.

Kurakin, A., Chien, S., Song, S., Geambasu, R., Terzis, A., and Thakurta, A. Toward training at imagenet scale with differential privacy. *arXiv preprint arXiv:2201.12328*, 2022.

Lee, J. and Kifer, D. Scaling up differentially private deep learning with fast per-example gradient clipping. *arXiv preprint arXiv:2009.03106*, 2020.

Li, X., Tramer, F., Liang, P., and Hashimoto, T. Large language models can be strong differentially private learners. *arXiv preprint arXiv:2110.05679*, 2021.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Mahabadi, R. K., Henderson, J., and Ruder, S. Compacter: Efficient low-rank hypercomplex adapter layers. *arXiv preprint arXiv:2106.04647*, 2021.

Marcel, S. and Rodriguez, Y. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*, pp. 1485–1488, 2010.

Mehta, H., Thakurta, A., Kurakin, A., and Cutkosky, A. Large scale transfer learning for differentially private image classification. *arXiv preprint arXiv:2205.02973*, 2022.

Mironov, I. Rényi differential privacy. In *2017 IEEE 30th computer security foundations symposium (CSF)*, pp. 263–275. IEEE, 2017.

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

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Rochette, G., Manoel, A., and Tramel, E. W. Efficient per-example gradient computations in convolutional neural networks. *arXiv preprint arXiv:1912.06015*, 2019.

Shokri, R., Stronati, M., Song, C., and Shmatikov, V. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pp. 3–18. IEEE, 2017.

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

Subramani, P., Vadivelu, N., and Kamath, G. Enabling fast differentially private sgd via just-in-time compilation and vectorization. *Advances in Neural Information Processing Systems*, 34, 2021.

Tensorflow. Tensorflow/privacy: Library for training machine learning models with privacy for training data. URL `https://github.com/tensorflow/privacy`.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Tramer, F. and Boneh, D. Differentially private learning needs better features (or much more data). *arXiv preprint arXiv:2011.11660*, 2020.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. 2019. In the Proceedings of ICLR.

Welbl, J., Stenetorp, P., and Riedel, S. Constructing datasets for multi-hop reading comprehension across documents. *Transactions of the Association for Computational Linguistics*, 6:287–302, 2018.

Wightman, R. Pytorch image models. `https://github.com/rwightman/pytorch-image-models`, 2019.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

Yousefpour, A., Shilov, I., Sablayrolles, A., Testuggine, D., Prasad, K., Malek, M., Nguyen, J., Ghosh, S., Bharadwaj, A., Zhao, J., Cormode, G., and Mironov, I. Opacus: User-friendly differential privacy library in PyTorch. *arXiv preprint arXiv:2109.12298*, 2021.

Yu, D., Naik, S., Backurs, A., Gopi, S., Inan, H. A., Kamath, G., Kulkarni, J., Lee, Y. T., Manoel, A., Wutschitz, L., et al. Differentially private fine-tuning of language models. *arXiv preprint arXiv:2110.06500*, 2021.

Zhu, Y., Dong, J., and Wang, Y.-X. Optimal accounting of differential privacy via characteristic function. *arXiv preprint arXiv:2106.08567*, 2021.

A Background

A.1 Differential privacy

We formally introduce the differential privacy (DP).

Definition A.1 ((Dwork et al., 2006))**.** A randomized algorithm M is (ε, δ)-differentially private (DP) if for any two neighboring[5] datasets S, S', and for any event E,

$$\mathbb{P}[M(S) \in E] \leqslant \mathrm{e}^{\varepsilon} \mathbb{P}\left[M\left(S'\right) \in E\right]+\delta. \tag{3}$$

Clearly, stronger DP (smaller ϵ, δ) indicates the higher difficulty for privacy attackers to extract information from the training data.

DP can be achieved by adding Gaussian noise to a bounded-sensitivity function (see Theorem A.1 of (Dwork et al., 2014)). In deep learning, this function is the sum of per-sample gradients $\sum \boldsymbol{g}_i$ and the bounded sensitivity is R (that is guaranteed through the gradient clipping after which the per-sample gradient norm is at most R). Note that the Gaussian noise magnitude is proportional to the sensitivity: $\sigma_{\text{DP}} = \sigma R$ in Equation (1), and $\epsilon(\delta)$ only depends on σ, not on R. The derivation from (σ, T, p) in Algorithm 1 to ϵ can be done through various methods in Section 1.3.

A.2 Computation graph

We elaborate on the computation graph presented in Figure 1. For DP and the standard training, the forward pass is the same: we pass through the layers

$$\boldsymbol{a}_{(1)} \rightarrow \boldsymbol{s}_{(1)} \rightarrow \boldsymbol{a}_{(2)} \rightarrow \boldsymbol{s}_{(2)} \rightarrow \cdots \boldsymbol{a}_{(L)} \rightarrow \boldsymbol{s}_{(L)}$$

For the backward propagation, there are two sub-processes:

1. the computation of **output gradient** *for all* layers,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(1)}} \leftarrow \cdots \leftarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l+1)}} \mathbf{W}_{(l+1)} \circ \mathrm{ReLU}'(\boldsymbol{s}_{(l)}) \leftarrow \cdots \leftarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(L)}},$$

   i.e. the output gradient meets with the weight $\mathbf{W}$;

2. the computation of **parameter gradient** *only for trainable* parameters,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}^{\top} \frac{\partial \boldsymbol{s}_{(l)}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}^{\top} \boldsymbol{a}_{(l)},$$

   i.e. the output gradient meets with the activation tensor $\boldsymbol{a}$.

Note that foward pass, output gradient, and parameter gradient have the same time complexity of $2BTM$ (B being the batch size, T being the feature dimension, e.g. the sequence length in texts, and M being the model size).

For example, GhostClip (Li et al., 2021) and MixGhostClip (Bu et al., 2022a), which use one forward pass and double backward propagation, have a time complexity of $10BTM + O(BT^2)$, while the standard training which uses one forward pass and a single backward propagation has a time complexity of $6BTM$.

B Complexity analysis for one layer

Let us consider a layer without bias term for simplicity:

$$\boldsymbol{s} = \boldsymbol{a}\mathbf{W} \tag{4}$$

[5] S' is a neighbor of S if one can obtain S' by adding or removing one data point from S.

where $\boldsymbol{s} \in \mathbb{R}^{B\times T\times p}$ is the output or the pre-activation, $\boldsymbol{a} \in \mathbb{R}^{B\times T\times d}$ is the input or the post-activation of previous layer, and $\mathbf{W} \in \mathbb{R}^{d\times p}$ is the weight matrix. In a linear layer, d is the input dimension of the hidden feature, p is the output dimension of the hidden feature, and T is the sequence length (or 1 if the data are non-sequential). In a convolution layer, d is the product of the input channels and kernel sizes, p is the output channels, T is the height times width of the hidden representation.

We now break down the time and space complexities for each operation in the training. Notice that we focus on major complexities, e.g. ignoring cubic terms like BTp when higher order terms like $BTpd$ or BT^2p exist.

B.1 Forward pass

The complexity of forward pass is incurred by the standard matrix multiplication $\boldsymbol{s} = \boldsymbol{a}\mathbf{W}$. Since $\boldsymbol{a} \in \mathbb{R}^{B\times T\times d}$ and $\mathbf{W} \in \mathbb{R}^{d\times p}$, the time complexity is $2BTpd$ and the space complexity is $BTp + pd$.

B.2 Back-propagation: output gradient

The complexity to compute the output gradient is incurred by the chain rule: for a single sample,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l-1),i}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}}_{\mathbb{R}^{T\times p}} \underbrace{\mathbf{W}_{(l)}^\top}_{\mathbb{R}^{p\times d}} \circ \underbrace{\phi'(\boldsymbol{s}_{(l-1),i})}_{\mathbb{R}^{T\times d}}$$

where ϕ is the non-linear activation function. We compute the matrix multiplication $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\mathbf{W}_{(l)}$ first, with time complexity $2BTpd$ and space complexity $pd + BTd + BTp$. Then the elementwise product uses time complexity $2BTd$ and space complexity BTd.

B.3 Back-propagation: parameter gradient

This module could represent different operations in different DP implementations. In the first back-propagation of GhostClip and the only back-propagation of Opacus, it computes $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \sum_i \mathcal{L}_i}{\partial \mathbf{W}}$; in the second back-propagation of Ghost/FastGradClip/BK, it computes the clipped gradient $\frac{\partial \sum_i C_i \mathcal{L}_i}{\partial \mathbf{W}}$. Regardless of the cases, the operation always takes the same format as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \underbrace{\boldsymbol{a}}_{\mathbb{R}^{B\times T\times d}}{}^\top \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}}_{\mathbb{R}^{B\times T\times p}} .$$

In contrast to the per-sample gradient instantiation, this operation is a tensor multiplication instead of many matrix multiplication, and the output is a single pair of gradient $\mathbb{R}^{d\times p}$ instead of many per-sample gradients.

This tensor multiplication has time complexity $2BTpd$ and space complexity pd unless all per-sample gradients are stored.

B.4 Ghost norm

Ghost norm is the operation taking $\boldsymbol{a}_i$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}$ as the input and outputs the per-sample gradient norm. According to Equation (2) and Appendix C.3 of (Bu et al., 2022b), this operation computes $\boldsymbol{a}_i\boldsymbol{a}_i^\top$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}^\top$, taking the time complexity $2BT^2d$ and $2BT^2p$ respectively, and the space complexity BT^2 for each variable. Hence total time complexity is $2BT^2(p + d)$ and total space complexity is $2BT^2$.

B.4.1 Per-sample gradient instantiation

Alternatively, one can instantiate the per-sample gradients and then compute their norms. This is different than the computation of parameter gradient in the back-propagation: that computation is an efficient tensor multiplication while this operation consists of B matrix multiplication.

$$\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}} = \underbrace{\boldsymbol{a}_i}_{\mathbb{R}^{T\times d}} \underbrace{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_i}}_{\mathbb{R}^{T\times p}}{}^\top \text{ for } i \in [B].$$

This operation has time complexity $2BTpd$ and space complexity Bpd to store all per-sample gradients. Computing the norms is cheap enough to be neglected.

B.5 Weighted sum of per-sample gradient

This operation simply takes per-sample clipping factor $C_i \in \mathbb{R}$ and $\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}} \in \mathbb{R}^{B\times d\times p}$ as the input, and outputs the clipped gradient $\mathbb{R}^{d\times p}$ as a weighted sum $\sum_i C_i \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}$. The time complexity is $2Bpd$ and the space complexity is 0 since the summation happens in place.

In contrast to double back-propagation, which indirectly derives the clipped gradient by differentiating the reweighted loss $\sum_i C_i \mathcal{L}_i$ at a cost of $O(BTpd)$, this operation directly computes the clipped gradient under almost no time complexity. Noticeably, this is only possible if per-sample gradients are readily instantiated and stored.

C Line-by-line comparison between different implementations

C.1 BK v.s. GhostClip

Algorithm 2 DP optimizer with BK or GhostClip

Parameters: l-th layer weights $\mathbf{W}_{(l)}$, number of layers L, noise level σ.

 # forward pass
for layer $l \in 1, 2, \cdots, L$ **do**
  Get $\{\boldsymbol{a}_{(l),i}\}$
 # backward propagation with loss $\mathcal{L} = \sum_i \mathcal{L}_i$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Get output gradient $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
  Compute per-sample gradient norm: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2 = \text{vec}(\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}) \cdot \text{vec}(\boldsymbol{a}_{(l),i}^\top \boldsymbol{a}_{(l),i})$
  Compute non-private gradient: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \boldsymbol{a}_{(l)}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
Aggregate gradient norm across all layers: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F^2 = \sum_l \|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
Compute clipping factor: $C_i = C(\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F; R)$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \text{diag}(\mathbf{C}) \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  # 2nd backward propagation with loss $\mathcal{L} = \sum_i C_i \mathcal{L}_i$
  Get output gradient $\{\frac{\partial \sum C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l),i}}\}$
  Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \frac{\partial \sum C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l)}}$
  Delete $\{\boldsymbol{a}_{(l),i}\}$, $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$, $\{\frac{\partial \sum C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l),i}}\}$
Add Gaussian noise $\hat{\mathbf{G}} = \mathbf{G} + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
Apply SGD/Adam/LAMB with the private gradient $\hat{\mathbf{G}}$ on $\mathbf{W}$

C.2 BK v.s. Opacus

Algorithm 3 DP optimizer with BK or Opacus

Parameters: l-th layer's weights $\mathbf{W}_{(l),t}$, number of layers L, noise scale σ.

for layer $l \in 1, 2, \cdots, L$ **do**
  Get $\{\boldsymbol{a}_{(l),i}\}$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Get output gradient $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
  Compute per-sample gradient norm: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2 = \text{vec}(\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}) \cdot \text{vec}(\boldsymbol{a}_{(l),i}^\top \boldsymbol{a}_{(l),i})$
  Compute non-private gradient: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \boldsymbol{a}_{(l)}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  Compute per-sample gradients: $\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}} = \boldsymbol{a}_{(l),i}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}$ and gradient norms $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
  Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
Aggregate gradient norm across all layers: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F^2 = \sum_l \|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
Compute clipping factor: $C_i = C(\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F; R)$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \text{diag}(\mathbf{C}) \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  Compute sum of clipped gradients $\mathbf{G}_l = \sum C_i \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}$
  Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$, $\{\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}}\}$
Add Gaussian noise $\hat{\mathbf{G}} = \mathbf{G} + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
Apply SGD/Adam/LAMB with the private gradient $\hat{\mathbf{G}}$ on $\mathbf{W}$

C.3 BK v.s. standard (non-DP)

Algorithm 4 DP optimizer with BK or Standard optimizer

Parameters: l-th layer's weights $\mathbf{W}_{(l),t}$, number of layers L, noise scale σ.

for layer $l \in 1, 2, \cdots, L$ **do**
  Get $\{\boldsymbol{a}_{(l),i}\}$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Get output gradient $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
  Compute per-sample gradient norm: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2 = \text{vec}(\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}) \cdot \text{vec}(\boldsymbol{a}_{(l),i}^\top \boldsymbol{a}_{(l),i})$
  Compute non-private gradient: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \boldsymbol{a}_{(l)}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
Aggregate gradient norm across all layers: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F^2 = \sum_l \|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
Compute clipping factor: $C_i = C(\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F; R)$
for layer $l \in L, L-1, \cdots, 1$ **do**
  Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \text{diag}(\mathbf{C}) \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
  Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
Add Gaussian noise $\hat{\mathbf{G}} = \mathbf{G} + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
Apply SGD/Adam/LAMB with $\hat{\mathbf{G}}$ or $\mathbf{G}$ on $\mathbf{W}$

C.4 BK (base) v.s. hybrid BK

Algorithm 5 DP optimizer with BK, BK-MixGhostClip or BK-MixOpt

Parameters: l-th layer's weights $\mathbf{W}_{(l)}$, number of layers L, noise scale σ.

 # forward pass
for layer $l \in 1, 2, \cdots, L$ **do**
    Get $\{\boldsymbol{a}_{(l),i}\}$
 # backward propagation with loss $\mathcal{L} = \sum_i \mathcal{L}_i$
for layer $l \in L, L-1, \cdots, 1$ **do**
    Get output gradient $\{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}$
    if (MixGhostClip or MixOpt) and $2T_{(l)}^2 > p_{(l)} d_{(l)}$ **then**
        Compute per-sample gradients: $\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}} = \boldsymbol{a}_{(l),i}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}$ and gradient norms $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
    else
        Compute per-sample gradient norm: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2 = \text{vec}(\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}{}^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}) \cdot \text{vec}(\boldsymbol{a}_{(l),i}^\top \boldsymbol{a}_{(l),i})$
Aggregate gradient norm across all layers: $\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F^2 = \sum_l \|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\|_F^2$
Compute clipping factor: $C_i = C(\|\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}\|_F; R)$
for layer $l \in L, L-1, \cdots, 1$ **do**
    if MixOpt and $2T_{(l)}^2 > p_{(l)} d_{(l)}$ **then**
        Compute weighted gradients $\mathbf{G}_l = \sum C_i \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}$
    else
        Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \text{diag}(\mathbf{C}) \frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l)}}$
    Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}, \{\frac{\partial \mathcal{L}_i}{\partial \mathbf{W}_{(l)}}\}$
Add Gaussian noise $\hat{\mathbf{G}} = \mathbf{G} + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
Apply SGD/Adam/LAMB with the private gradient $\hat{\mathbf{G}}$ on $\mathbf{W}$

D Codebase README

Here we describe some designs in our codebase for BK algorithms.

D.1 Supported layers

- Linear: Ghost norm or per-sample gradient instantiation
- Embedding: Ghost norm
- Conv1d & Conv2d & Conv3d: Ghost or per-sample gradient instantiation
- GroupNorm & LayerNorm & InstanceNorm: per-sample gradient instantiation

D.2 Instruction of implementation

In this section, we will discuss the specific designs and tricks for our book-keeping technique. We illustrate through Pytorch automatic differentiation package, known as `torch.autograd` or simply `autograd`[6]. It has two high-level operators, `autograd.backward` (which is the major component of the commonly used `loss.backward()`) and `autograd.grad`. We denote the model parameters as `param`.

On all trainable layers, i.e. layers with at least one trainable parameter such that `param.requires_grad=True`, the operator `autograd.backward` does three things, 1. compute the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$ for this layer; 2. compute the parameter gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ or $\frac{\partial \mathcal{L}}{\partial \mathbf{b}}$; 3. store the parameter gradient to `param.grad` attribute.

[6] See `https://pytorch.org/docs/stable/autograd.html` for an official introduction.

In contrast, `autograd.grad` returns but does not store the parameter gradient in step 3. However, `autograd.grad` still computes the parameter gradient in step 2 (or 2b) unnecessarily.

Therefore the key idea is to only compute the output gradient without computing the parameter gradient. This goal can be achieved by

1. registering the Pytorch backward hooks, which have free access to the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$, to store this output gradient for 2a (Line 9 of Algorithm 1);
2. setting all parameters to not require gradients, through `requires_grad=False`.

D.3 Work-around: origin parameters

Unfortunately, the back-propagation will not be executed if all parameters are set to not require gradients, since the computation graph needs to be created at least on some trainable parameters. Therefore, while the above methodology is certainly implementable through mild modification on the low level (like CUDA kernel), we provide a lightweight work-around in Pytorch.

To make sure that the back-propagation indeed propagates through all trainable parameters, we set `param.requires_grad=True` on and only on the ancestor parameter nodes of all output nodes, termed as the **origin parameters**. Specifically, we define the origin parameters as the *subset of parameter nodes, whose descendant nodes cover all the output nodes*. This is visualized in Figure 8 for a 3-layer MLP, using the same symbols as Figure 1.

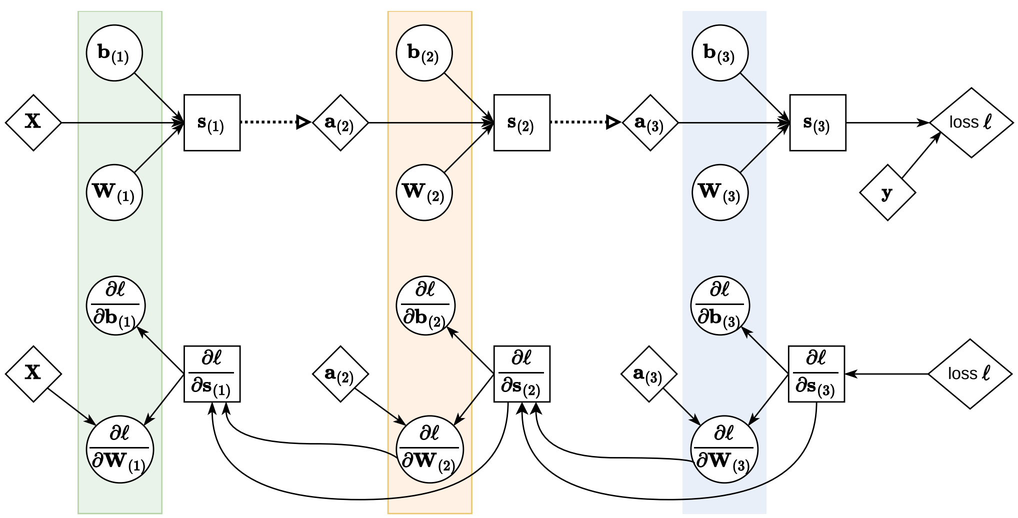

Figure 8. Forward pass (upper panel) and back-propagation (lower panel) of a 3-layer MLP.

Here, $\boldsymbol{s}_{(i)}$ are the output nodes (in squares) from the trainable layers. The ancestor parameter nodes (in circles) of $\boldsymbol{s}_{(3)}$ are $\{\mathbf{b}_{(3)}, \mathbf{b}_{(2)}, \mathbf{b}_{(1)}, \mathbf{W}_{(3)}, \mathbf{W}_{(2)}, \mathbf{W}_{(1)}\}$, those of $\boldsymbol{s}_{(2)}$ are $\{\mathbf{b}_{(2)}, \mathbf{b}_{(1)}, \mathbf{W}_{(2)}, \mathbf{W}_{(1)}\}$, and those of $\boldsymbol{s}_{(1)}$ are $\{\mathbf{b}_{(1)}, \mathbf{W}_{(1)}\}$. Therefore, subsets including but not limited to $\{\mathbf{b}_{(3)}, \mathbf{b}_{(2)}, \mathbf{b}_{(1)}, \mathbf{W}_{(3)}, \mathbf{W}_{(2)}, \mathbf{W}_{(1)}\}$, $\{\mathbf{b}_{(1)}, \mathbf{W}_{(1)}\}$, and $\{\mathbf{b}_{(1)}\}$ are qualified as the origin parameters, since their descendants cover all output nodes. In fact, the smallest subsets are $\{\mathbf{b}_{(1)}\}$ or $\{\mathbf{W}_{(1)}\}$, and either can serve as the optimal origin parameters.

Remark D.1. The origin parameters are usually within the embedding layer in language models and transformers, or within the first convolution layer in vision models. Since the origin parameters only constitute a small fraction of all trainable parameters (fewer than the parameters in the first layer) in deep neural networks (with hundreds of layers), the computational overhead wasted on the regular gradient of origin parameters is negligible.

Remark D.2. Since we will waste the computation of regular gradient $\frac{\partial \mathcal{L}}{\partial \texttt{origin\_parameters}}$, it is preferred to use the bias over the weight for minimum waste whenever possible. We note that sometimes the first layer contains no bias term. For example, the embedding layer by `torch.nn.Embedding` has no bias by design, and so do all convolution layers in

ResNets from torchvision (Marcel & Rodriguez, 2010), with reasons discussed at Section 3.2 of (Ioffe & Szegedy, 2015), which generalizes to all batch-normalized CNN if the normalization is applied before the activation function.

In summary, we drive the back-propagation without computing the regular parameter gradient $\frac{\partial \sum_i \mathcal{L}_i}{\partial \mathbf{W}}$ (by setting `param.requires_grad=False`), and use Pytorch backward hooks to access and store the output gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}}$.

	non-DP training		DP training (Book-Keeping)		
	trainable param	non-trainable param	trainable param (origin param)	trainable param (not origin param)	non-trainable param
register hook	✗	✗	✓	✓	✗
`param.requires_grad`	✓	✗	✓	✗	✗

Table 6. Origin parameter trick and implementation details.

D.4 How to use BK codebase

With a few lines of code, it is easy to use our BK codebase to change the standard training to the DP training. All you need to do is to declare a privacy engine and attach it to the optimizer.

```
from BK import PrivacyEngine
from transformers import AutoModel

model = AutoModel.from_pretrained('roberta-base')

optimizer = torch.optim.Adam(params=model.parameters())

privacy_engine = PrivacyEngine(
    model,batch_size=256,sample_size=50000,
    epochs=3,target_epsilon=3,clipping_mode='MixOpt')
privacy_engine.attach(optimizer)

# Same training procedure, e.g. data loading, forward pass, logits...
loss = torch.nn.functional.cross_entropy(logits, labels)
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

Notice that if `clipping_mode` is set to default, then BK (base) is implemented; if `clipping_mode=='MixGhostClip'`, then BK-MixGhostClip is implemented; if `clipping_mode=='MixOpt'`, then BK-MixOpt is implemented.

We also allow the gradient accumulation in the same way as non-private training.

E Applicability of BK algorithm

E.1 Applying BK to full fine-tuning

We experiment with numerous vision and language models to show the strong applicability of BK. Notice that the ghost norm trick only applies on weight parameters and in the generalized linear layers, i.e. embedding/convolutional/linear. The vision models are imported from Pytorch Image Models library (Wightman, 2019) and the language models are imported from Hugging Face Transformers library (Wolf et al., 2020)[7].

[7]In Transformers library, layers with class name 'Conv1D' is actually a linear layer, different from 1D convolution `torch.nn.Conv1d`.

E.2 Applying BK to parameter efficient fine-tuning

We demonstrate that BK (base and hybrid) can be applied to DP LoRA and DP Adapter, where the rank r is usually 16-1024. For the ease of presentation, we describe the BK base, similarly to Algorithm 1.

Adapter An adapter module is injected after a linear layer:

$$A(x) = \tau(xD)U + x$$

where $x \in \mathbb{R}^{B\times T\times p}, D \in \mathbb{R}^{p\times r}, U \in \mathbb{R}^{r\times p}$. We decompose the module A into two sub-modules:

- $x \to xD := u$, activation x, output grad $\frac{\partial \mathcal{L}}{\partial u}$
- $\tau(u) \to \tau U := v$, activation $\tau(xD)$, output grad $\frac{\partial \mathcal{L}}{\partial v}$

Hence BK can be implemented as follows.

1. Get activation tensors x and $\tau(xD)$ by Pytorch forward hook
2. Get output gradients $\{\frac{\partial \mathcal{L}}{\partial xD}\}$ and $\{\frac{\partial \mathcal{L}}{\partial \tau U}\}$ by Pytorch backward hook
3. Compute per-example gradient norm $\|\frac{\partial \mathcal{L}_i}{\partial D}\|_F^2$ and $\|\frac{\partial \mathcal{L}_i}{\partial U}\|_F^2$ by ghost norm trick
4. Aggregate gradient norm across all layers: $\|\frac{\partial \mathcal{L}_i}{\partial D}\|_F^2 + \|\frac{\partial \mathcal{L}_i}{\partial U}\|_F^2$
5. Compute clipping factor C_i
6. Compute sum of clipped gradients $\mathbf{G}_D = x^\top \text{diag}(C_1, C_2, \cdots)\frac{\partial \mathcal{L}}{\partial xD}$ and $\mathbf{G}_U = \tau^\top \text{diag}(C_1, C_2, \cdots)\frac{\partial \mathcal{L}}{\partial \tau U}$
7. Add Gaussian noise $\hat{\mathbf{G}}_D = \mathbf{G}_D + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$ and $\hat{\mathbf{G}}_U = \mathbf{G}_U + \sigma R \cdot \mathcal{N}(0, \mathbf{I})$
8. Apply SGD/Adam/LAMB with the private gradient $\hat{\mathbf{G}}_D$ on D and $\hat{\mathbf{G}}_U$ on U

Existing implementation of DP Adapter [8] uses the per-sample gradient instantiation as in Opacus. It is not hard to see that the layerwise space overhead (in addition to forward pass and output gradient) is $2Bpr$ and the time overhead is $4BTpr$ (c.f. Table 3 4). With the BK implementation, the space overhead is $4BT^2$ and the time overhead is $4BT^2(p+r)$ (c.f. Table 3 3).

LoRA LoRA modifies

$$A(x) = x(W + LR) = xW + xLR$$

where $x \in \mathbb{R}^{B\times T\times d}, W \in \mathbb{R}^{d\times p}, L \in \mathbb{R}^{d\times r}, R \in \mathbb{R}^{r\times p}$. We decompose the module A into two sub-modules:

- $x \to xL := u$, activation x, output grad $\frac{\partial \mathcal{L}}{\partial u}$
- $u \to uR := v$, activation xL, output grad $\frac{\partial \mathcal{L}}{\partial v}$

Hence BK can be implemented on each sub-module, similar to the DP Adapter.

Existing implementation of DP LoRA [9] uses the per-sample gradient instantiation as in Opacus. It is not hard to see that the layerwise space overhead (in addition to forward pass and output gradient) is $Br(p+d)$ and the time overhead is $2BTr(p+d)$ (c.f. Table 3 4). With the BK implementation, the space overhead is $4BT^2$ and the time overhead is $2BT^2(p+d+2r)$ (c.f. Table 3 3).

[8] https://github.com/huseyinatahaninan/Differentially-Private-Fine-tuning-of-Language-Models/tree/main/Language-Understanding-RoBERTa/bert_adapter

[9] https://github.com/huseyinatahaninan/Differentially-Private-Fine-tuning-of-Language-Models/tree/main/Language-Understanding-RoBERTa/bert_lora

Model	# param in generalized linear layers		# param in other layers	% applicable to BK
	weight	bias	weight+bias	
ResNet18	11.7M	1000	9600	99.9%
ResNet34	21.8M	1000	17024	99.9%
ResNet50	25.5M	1000	53120	99.8%
ResNet101	44.4M	1000	105344	99.8%
ResNet152	60.2M	1000	151424	99.7%
DenseNet121	7.9M	1000	83648	98.9%
DenseNet161	28.5M	1000	219936	99.2%
DenseNet201	19.8M	1000	229056	98.9%
Wide ResNet50	68.8M	1000	68224	99.9%
Wide ResNet101	126.7M	1000	137856	99.9%
vit_tiny_patch16_224	5.6M	21928	9600	99.4%
vit_small_patch16_224	21.9M	42856	19200	99.7%
vit_base_patch16_224	86.3M	84712	38400	99.9%
vit_large_patch16_224	303.8M	223208	100352	99.9%
crossvit_tiny_240	6.9M	30800	16128	99.3%
crossvit_small_240	26.6M	59600	32256	99.7%
crossvit_base_240	104.5M	117200	64512	99.8%
convnext_small	50.1M	83656	30144	99.8%
convnext_base	88.4M	111208	40192	99.8%
convnext_large	197.5M	166312	60288	99.9%
deit_tiny_patch16_224	5.6M	21928	9600	99.4%
deit_small_patch16_224	21.9M	42856	19200	99.7%
deit_base_patch16_224	86.3M	84712	38400	99.9%
beit_base_patch16_224	86.3M	57064	38400	99.9%
beit_large_patch16_224	303.8M	149480	100352	99.9%
roberta-base	124.5M	83712	38400	99.9%
roberta-large	355.0M	222208	100352	99.9%
distilroberta-base	82.1M	42240	19968	99.9%
bert-base-uncased	109.4M	83712	38400	99.9%
bert-large-uncased	334.8M	222208	100352	99.9%
bert-base-cased	108.2M	83712	38400	99.9%
bert-large-cased	333.3M	222208	100352	99.9%
longformer-base-4096	148.5M	111360	38400	99.9%
longformer-large-4096	434.2M	295936	100352	99.9%
t5-small	60.5M	0	16384	99.9%
t5-base	222.9M	0	47616	99.98%
t5-large	737.5M	0	124928	99.98%
long-t5-local-base	222.9M	0	47616	99.98%
long-t5-local-large	750.1M	0	124928	99.98%
long-t5-tglobal-base	222.9M	0	56832	99.97%
long-t5-tglobal-large	750.1M	0	149504	99.98%
gpt2	124.3M	82944	38400	99.9%
gpt2-medium	354.5M	221184	100352	99.9%
gpt2-large	773.4M	414720	186880	99.9%

Table 7. Models and the percentage of trainable parameters in generalized linear layers.

F Additional plots and tables

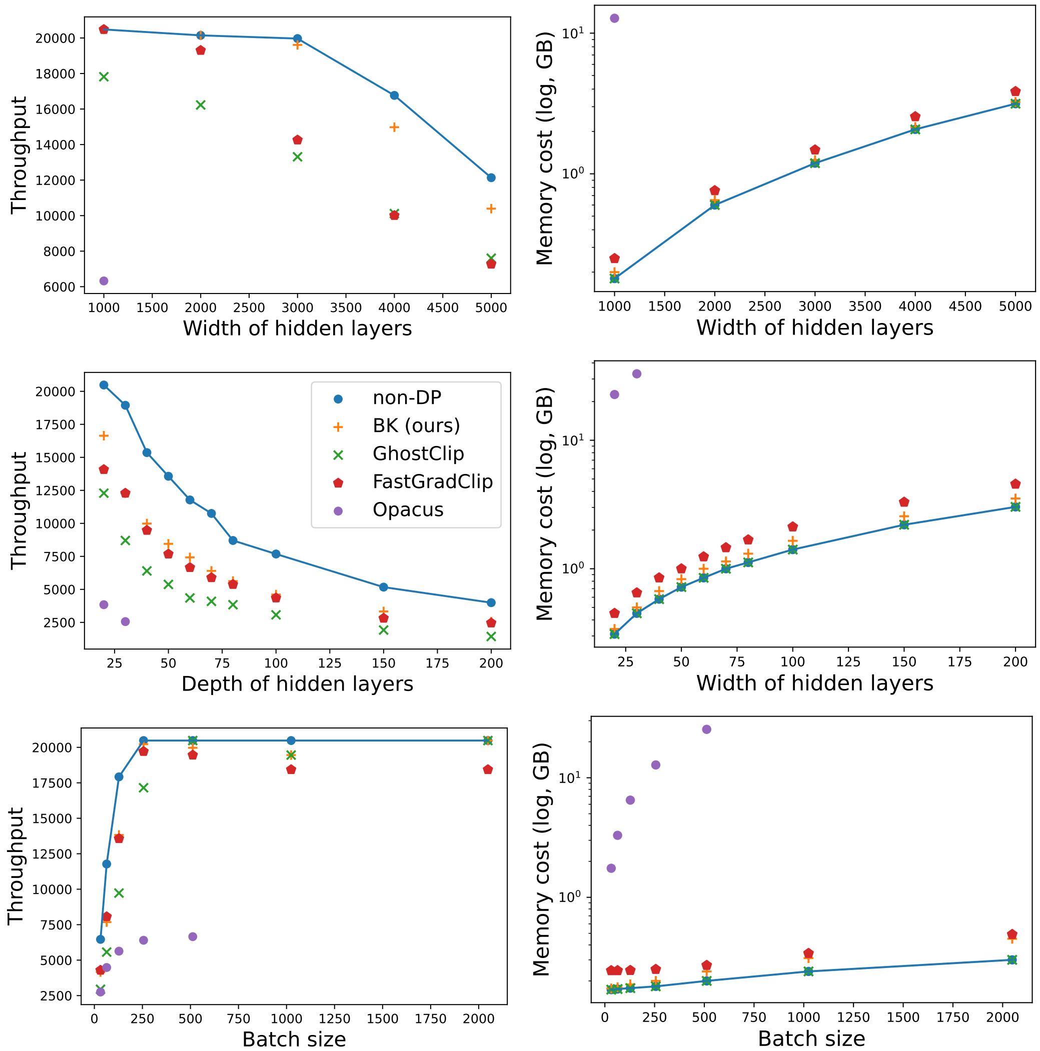

Figure 9. Ablation study of MLP on CIFAR10/CIFAR100 (images are flattened into vectors). Default model: 10 layers, width 1000, batch size 256.

	BK	Non-DP	GhostClip	Opacus
Time complexity	$6B\sum_l T_{(l)}p_{(l)}d_{(l)}$ $+2B\sum_l\Big(\mathbb{I}\{2T_{(l)}^2 < p_{(l)}d_{(l)}\}$ $\cdot T_{(l)}^2(p_{(l)}+d_{(l)})\Big)$	$6B\sum_l T_{(l)}p_{(l)}d_{(l)}$	$10B\sum_l T_{(l)}p_{(l)}d_{(l)}$ $+2B\sum_l T_{(l)}^2(p_{(l)}+d_{(l)})$	$8B\sum_l T_{(l)}p_{(l)}d_{(l)}$
RoBERTa-base	$15.3 * 10^{12}$	$13.1 * 10^{12}(0.86\times)$	$24.1 * 10^{12}(1.57\times)$	$17.5 * 10^{12}(1.14\times)$
RoBERTa-large	$52.3 * 10^{12}$	$46.5 * 10^{12}(0.89\times)$	$83.3 * 10^{12}(1.59\times)$	$62.0 * 10^{12}(1.18\times)$
ViT-base	$11.2 * 10^{12}$	$10.1 * 10^{12}(0.90\times)$	$18.0 * 10^{12}(1.60\times)$	$13.5 * 10^{12}(1.20\times)$
ViT-large	$38.8 * 10^{12}$	$35.8 * 10^{12}(0.92\times)$	$62.7 * 10^{12}(1.61\times)$	$47.7 * 10^{12}(1.23\times)$
BEiT-large	$29.1 * 10^{12}$	$26.9 * 10^{12}(0.92\times)$	$47.1 * 10^{12}(1.61\times)$	$35.8 * 10^{12}(1.23\times)$
GPT2-small	$7.7 * 10^{12}$	$7.5 * 10^{12}(0.96\times)$	$12.7 * 10^{12}(1.64\times)$	$10.0 * 10^{12}(1.28\times)$
GPT2-medium	$22.1 * 10^{12}$	$21.4 * 10^{12}(0.96\times)$	$36.2 * 10^{12}(1.64\times)$	$28.4 * 10^{12}(1.29\times)$
GPT2-large	$47.9 * 10^{12}$	$46.4 * 10^{12}(0.97\times)$	$78.8 * 10^{12}(1.65\times)$	$61.9 * 10^{12}(1.30\times)$
GPT2-small	$9.3 * 10^{13}$	$7.5 * 10^{13}(0.80\times)$	$15.5 * 10^{13}(1.66\times)$	$9.9 * 10^{12}(1.07\times)$
GPT2-medium	$28.2 * 10^{13}$	$21.4 * 10^{13}(0.76\times)$	$43.4 * 10^{13}(1.54\times)$	$28.4 * 10^{13}(1.01\times)$
GPT2-large	$59.4 * 10^{13}$	$46.4 * 10^{13}(0.79\times)$	$92.2 * 10^{13}(1.55\times)$	$61.9 * 10^{13}(1.04\times)$
Space complexity	$B\sum_l \min\{2T_{(l)}^2, p_{(l)}d_{(l)}\}$ $+B\sum_l T_{(l)}(3d_{(l)}+p_{(l)})$	$\sum_l p_{(l)}d_{(l)}$ $+B\sum_l T_{(l)}(3d_{(l)}+p_{(l)})$	$2B\sum_l T_{(l)}^2$ $+B\sum_l T_{(l)}(3d_{(l)}+p_{(l)})$	$B\sum_l p_{(l)}d_{(l)}$ $+B\sum_l T_{(l)}(3d_{(l)}+p_{(l)})$
RoBERTa-base	$5.3 * 10^9$	$4.5 * 10^9(0.84\times)$	$5.3 * 10^9(1.00\times)$	$16.7 * 10^9(3.17\times)$
RoBERTa-large	$13.3 * 10^9$	$11.8 * 10^9(0.88\times)$	$13.3 * 10^9(1.00\times)$	$46.9 * 10^9(3.52\times)$
ViT-base	$3.3 * 10^9$	$3.0 * 10^9(0.91\times)$	$3.3 * 10^9(1.00\times)$	$11.5 * 10^9(3.47\times)$
ViT-large	$8.5 * 10^9$	$8.1 * 10^9(0.95\times)$	$8.5 * 10^9(1.00\times)$	$38.1 * 10^9(4.46\times)$
BEiT-large	$6.4 * 10^9$	$6.1 * 10^9(0.95\times)$	$6.4 * 10^9(1.00\times)$	$28.6 * 10^9(4.46\times)$
GPT2-small	$1.7 * 10^9$	$1.6 * 10^9(0.94\times)$	$1.7 * 10^9(1.00\times)$	$14.0 * 10^9(8.19\times)$
GPT2-medium	$4.5 * 10^9$	$4.3 * 10^9(0.96\times)$	$4.5 * 10^9(1.00\times)$	$39.8 * 10^9(8.82\times)$
GPT2-large	$8.47 * 10^9$	$8.17 * 10^9(0.97\times)$	$8.47 * 10^9(1.00\times)$	$85.5 * 10^9(10.1\times)$
GPT2-small	$2.3 * 10^{10}$	$1.5 * 10^{10}(0.68\times)$	$2.5 * 10^{10}(1.10\times)$	$2.8 * 10^{10}(1.20\times)$
GPT2-medium	$5.7 * 10^{10}$	$4.0 * 10^{10}(0.70\times)$	$6.0 * 10^{10}(1.04\times)$	$7.6 * 10^{10}(1.32\times)$
GPT2-large	$10.1 * 10^{10}$	$7.5 * 10^{10}(0.75\times)$	$10.5 * 10^{10}(1.02\times)$	$15.2 * 10^{10}(1.48\times)$

Table 8. Time (upper half) and space (lower half) complexity of implementations ($B = 100$). For text classification, $T = 256$ and we use BK base ($\equiv$ BK-MixOpt). For vision transformers and ImageNet, $T = 224 \times 224$ and we use BK-MixOpt. For text generation (GPT2, which has token length limit as 1024), we use $T = 100$ in black or 1000 in light cyan. We mark the ratio of an algorithm's complexity to BK's inside the parenthesis. Note that neither per-sample gradient instantiation (Opacus) nor ghost norm trick (GhostClip) is satisfying when T is large, and they are dominated by BK-MixOpt.

Model	Algorithm	Maximum batch size	Time/Epoch	Maximum throughput	Speedup by BK
RoBERTa-large SST-2	BK (ours)	41	13:03	86	—
	Non-private	51	9:50	114	0.75×
	GhostClip	48	17:34	64	1.34×
	Opacus	16	22:30	50	1.72×
RoBERTa-large QNLI	BK (ours)	41	20:14	86	—
	Non-private	51	15:33	112	0.77×
	GhostClip	48	27:45	63	1.37×
	Opacus	16	35:03	50	1.73×
RoBERTa-large QQP	BK (ours)	41	70:04	87	—
	Non-private	51	53:42	113	0.77×
	GhostClip	48	95:09	64	1.36×
	Opacus	16	137:00	44	1.96×
RoBERTa-large MNLI	BK (ours)	41	77:07	85	—
	Non-private	51	58:02	113	0.75×
	GhostClip	48	103:30	63	1.34×
	Opacus	16	134:30	49	1.75×
GPT2	BK (ours)	149	2:13	316	—
	Non-private	157	1:47	393	0.80×
	GhostClip	156	2:54	242	1.31×
	Opacus	43	5:03	139	2.27×
GPT2-medium	BK (ours)	69	4:58	141	—
	Non-private	70	4:05	172	0.82×
	GhostClip	70	6:46	104	1.36×
	Opacus	15	14:22	49	2.88×
GPT2-large	BK (ours)	29	10:01	70	—
	Non-private	29	8:16	85	0.83×
	GhostClip	29	13:56	50	1.36×
	Opacus	5	44:05	16	4.41×
BEiT-large	BK (ours)	96	6:35	127	—
	Non-private	98	4:55	169	0.76×
	GhostClip	95	8:53	93	1.33×
	Opacus	5	4:12:00	3	38.3×

Table 9. Extension of Table 1. Note that CIFAR means both CIFAR10 and CIFAR100. Performance of GPT2 on E2E dataset (same setting as (Li et al., 2021; Bu et al., 2022b)).

Model	Mixed ghost norm (MGN)	Per-sample grad instantiation		Ghost norm	
	$\sum_l \min\{2T_{(l)}^2, p_{(l)}d_{(l)}\}$	$(\sum_l p_{(l)}d_{(l)}$; # param)	Saving by MGN	$(\sum_l 2T_{(l)}^2 = 2H_{\text{out}}^2 W_{\text{out}}^2)$	Saving by MGN
ResNet18	1.0M	11.5M	11.5×	399M	399×
ResNet34	2.3M	21.6M	9.4×	444M	194×
ResNet50	2.8M	22.7M	8.0×	528M	186×
ResNet101	6.8M	41.7M	6.2×	532M	79×
ResNet152	10.9	57.3M	5.3×	549M	51×
DenseNet121	4.1M	7.9M	1.9×	605M	147×
DenseNet161	9.0M	28.5M	3.2×	607M	67×
DenseNet201	7.0M	19.8M	2.8×	609M	87×
Wide ResNet50	5.6M	66.0M	11.7×	528M	93×
Wide ResNet101	9.6M	124.0M	13.0×	531M	56×
vit_tiny_patch16_224	3.3M	5.6M	1.7×	3.8M	1.1×
vit_small_patch16_224	3.8M	21.9M	5.8×	13.8M	1.0×
vit_base_patch16_224	3.8M	86.3M	22.7×	3.8M	1.0×
vit_large_patch16_224	7.5M	303.8M	40.4×	7.5M	1.0×
crossvit_tiny_240	4.0M	6.9M	1.7×	10.4M	2.6×
crossvit_small_240	5.9M	26.6M	4.5×	10.4M	1.8×
crossvit_base_240	8.7M	104.5M	12.1×	10.4M	1.2×
convnext_small	12.4M	50.1M	4.0×	214M	17×
convnext_base	14.3M	88.4M	6.2×	214M	15×
convnext_large	19.8M	197.5M	10.0×	214M	11×
deit_tiny_patch16_224	3.3M	5.6M	1.7×	3.8M	1.1×
deit_small_patch16_224	3.8M	21.9M	5.8×	3.8M	1.0×
deit_base_patch16_224	3.8M	86.3M	22.7×	3.8M	1.0×
beit_base_patch16_224	2.9M	86.3M	29.8×	2.9M	1.0×
beit_large_patch16_224	5.7M	303.8M	53.3×	5.7M	1.0×

Table 10. Space complexity of computing per-sample gradient norm, on ImageNet image (224×224). The saving by the mixed ghost norm, adopted in BK-MixGhostClip and BK-MixOpt, is substantial.

G Effect of hybridization: layerwise space complexity

We demonstrate the effect of hybridization (i.e. mixed ghost norm (Bu et al., 2022a)) on the computation of per-sample gradient norm. We consider the moderate feature dimension and the high feature dimension, respectively. We conclude that ghost norm trick (adopted in GhostClip and BK) is favored closer to the input layer, whereas the per-sample gradient instantiation (adopted in Opacus and FastGradClip) is favored closer to the output layer.

G.1 Effect by model achitecture ($T = 224 \times 224$)

Generally speaking, CNN can benefit from hybridization, but vision transformers may not (unless the feature dimension is high, see next section for BEiT).

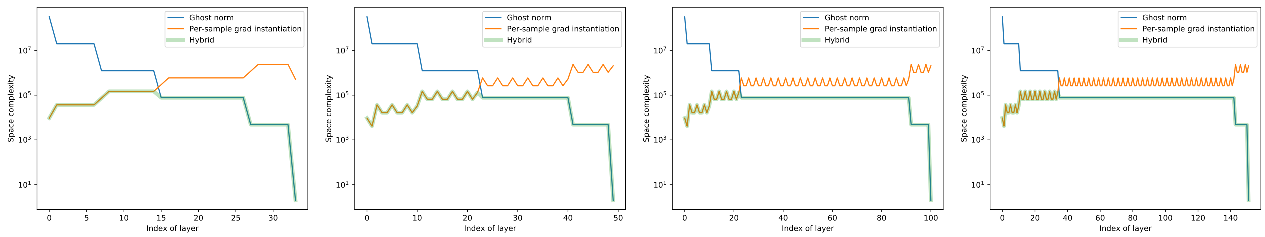

Figure 10. Layerwise space complexity of computing the per-sample gradient norm. Left to right: ResNet 34/50/101/152.

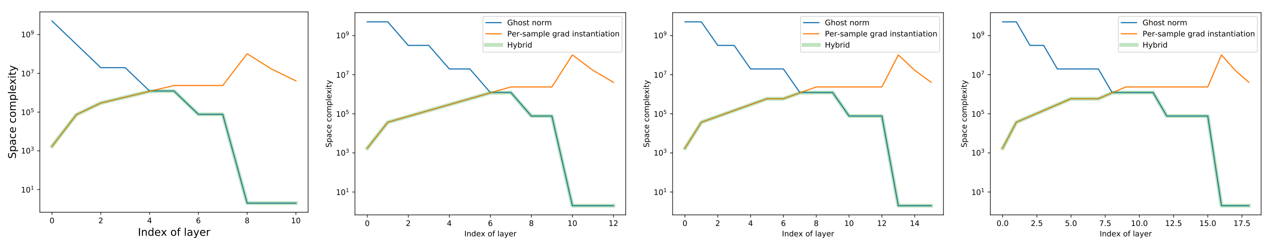

Figure 11. Layerwise space complexity of computing the per-sample gradient norm. Left to right: VGG 11/13/16/19.

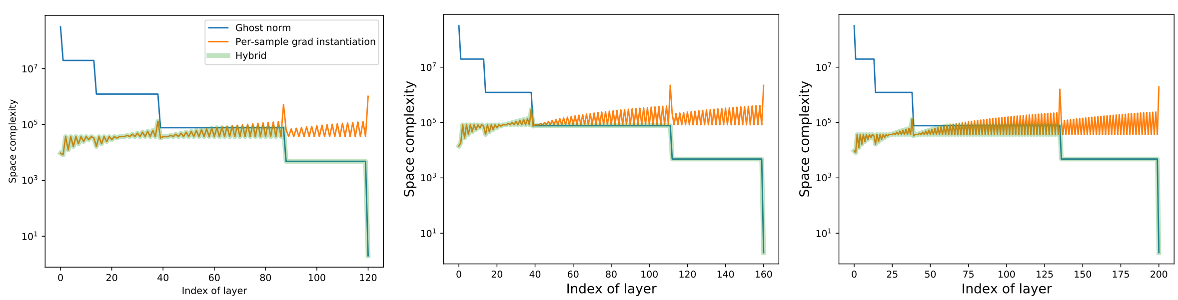

Figure 12. Layerwise space complexity of computing the per-sample gradient norm. Left to right: DenseNet 121/161/201.

G.2 Effect by feature dimension ($T = 32^2/224^2/512^2$)

Generally speaking, higher feature dimension requires a deeper threshold, after which the per-sample gradient instantiation is not preferred. That is, high dimensional data does not prefer ghost norm. This pattern even holds for vision transformers, on which MixGhostClip/BK-MixGhostClip is equivalent to GhostClip/BK for low feature dimension.

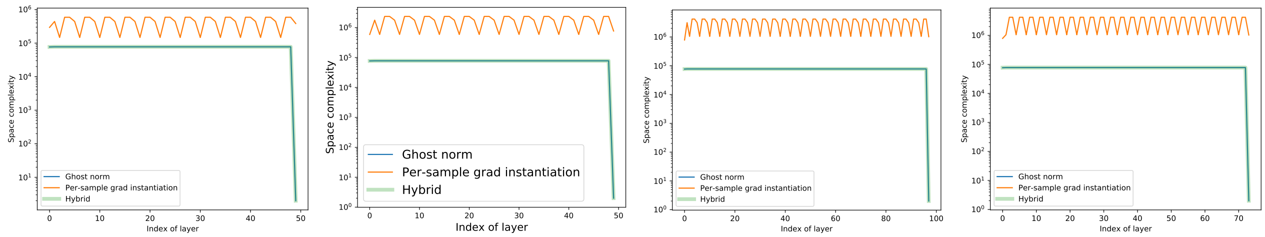

Figure 13. Layerwise space complexity of computing the per-sample gradient norm. Left to right: ViT small/base/large, and BEiT-large.

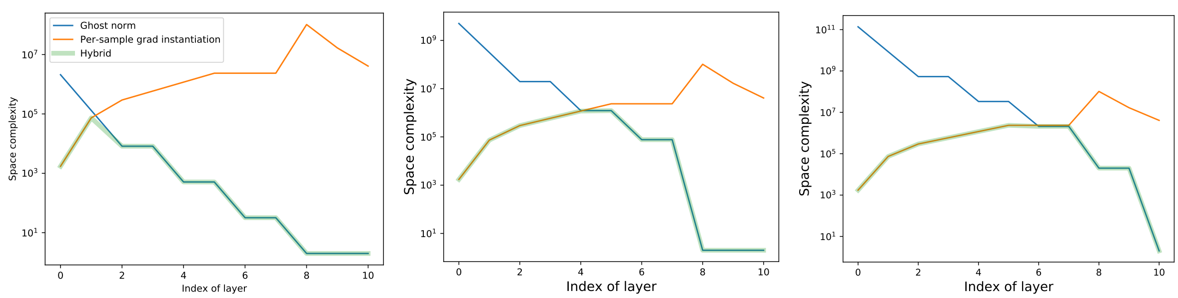

Figure 14. Layerwise space complexity of computing the per-sample gradient norm in VGG11.

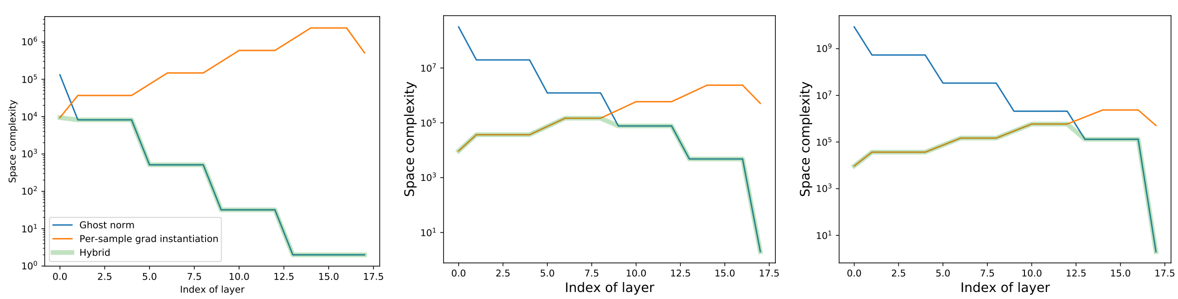

Figure 15. Layerwise space complexity of computing the per-sample gradient norm in ResNet18.

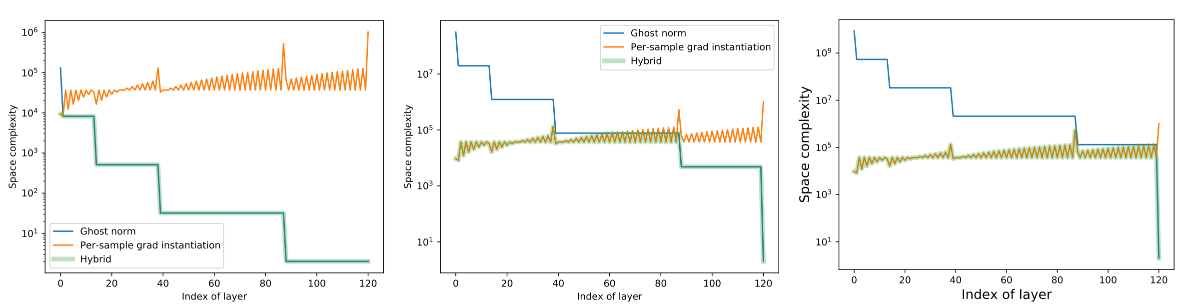

Figure 16. Layerwise space complexity of computing the per-sample gradient norm in DenseNet121.

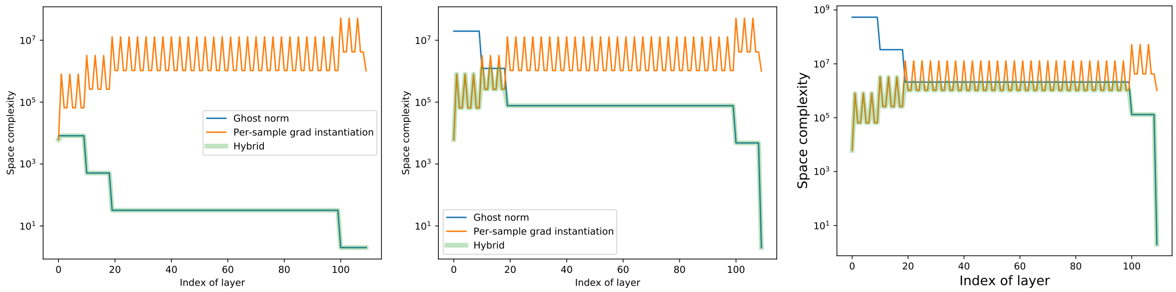

Figure 17. Layerwise space complexity of computing the per-sample gradient norm in ConvNeXT.

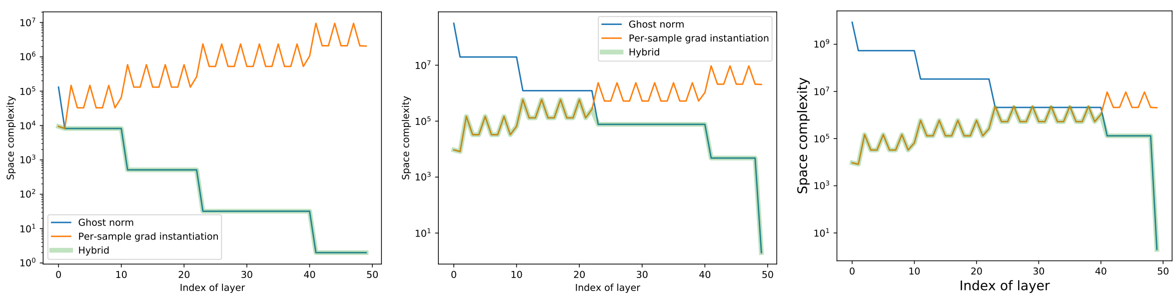

Figure 18. Layerwise space complexity of computing the per-sample gradient norm in Wide ResNet50.

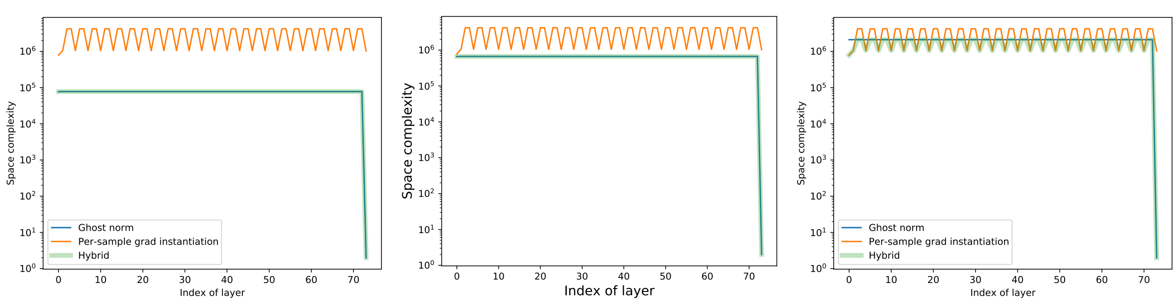

Figure 19. Layerwise space complexity of computing the per-sample gradient norm in BEiT-large.