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 (Dwork et al., 2014, Theorem A.1). 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

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

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:

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

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

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

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 (Bu et al., 2022b, Appendix C.3), 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.

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

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 $\{\frac{\partial \sum_i C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l),i}}\}$
    Compute sum of clipped gradients $\mathbf{G}_l = \boldsymbol{a}_{(l)}^\top \frac{\partial \sum_i C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l)}}$
    Delete $\{\boldsymbol{a}_{(l),i}\}, \{\frac{\partial \mathcal{L}}{\partial \boldsymbol{s}_{(l),i}}\}, \{\frac{\partial \sum_i C_i \mathcal{L}_i}{\partial \boldsymbol{s}_{(l),i}}\}$
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: Ghost or per-sample gradient instantiation
- GroupNorm&LayerNorm: 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, thus saving some memory cost. 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 7 for a 3-layer MLP, using the same symbols as Figure 1.

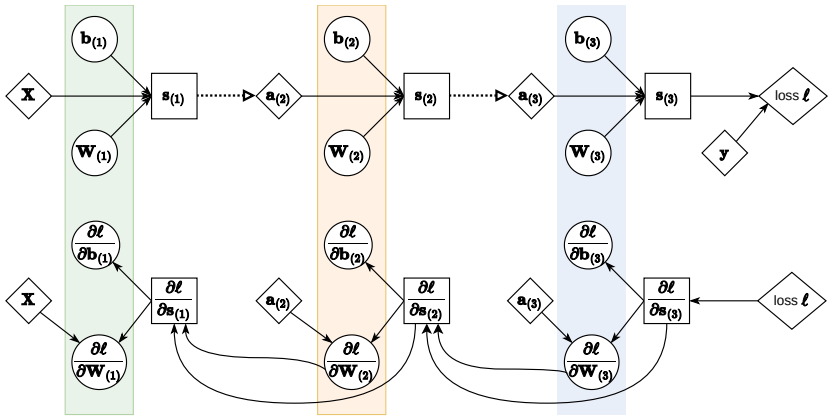

Figure 7: Forward pass (upper panel) and back-propagation (lower panel) of a 3-layer MLP.

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 (Ioffe & Szegedy, 2015, Section 3.2), which generalizes to all batch-normalized CNN if the normalization is applied before the activation function.

In summary, we use `torch.autograd.grad(loss, origin_parameters)` to 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. Note that we do not call `loss.backward()` as the back-propagation is implicitly called inside the `optimizer.step(loss=loss)` .

```
from BK import PrivacyEngine
from transformers import AutoModel
import torch.nn.functional as F

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

optimizer = torch.optim.Adam(params=model.parameters())
privacy_engine = PrivacyEngine(
    model,batch_size=256,sample_size=50000,max_grad_norm=0.1,
    epochs=3,target_epsilon=3,MixGhostClip=True,MixOpt=True)
privacy_engine.attach(optimizer)

# Same training procedure, e.g. data loading, forward pass, logits...
loss = F.cross_entropy(logits, labels)
optimizer.step(loss=loss)
```

Notice that if `MixGhostClip==False` and `MixOpt==False`, then BK (base) is implemented; if `MixGhostClip==True` and `MixOpt==False`, then BK-MixGhostClip is implemented; if `MixOpt==True`, then BK-MixOpt is implemented.

We also allows the gradient accumulation through `optimizer.virtual_step(loss=loss)`, similar to Opacus v0.15.0.

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`.

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.

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

F ADDITIONAL PLOTS AND TABLES

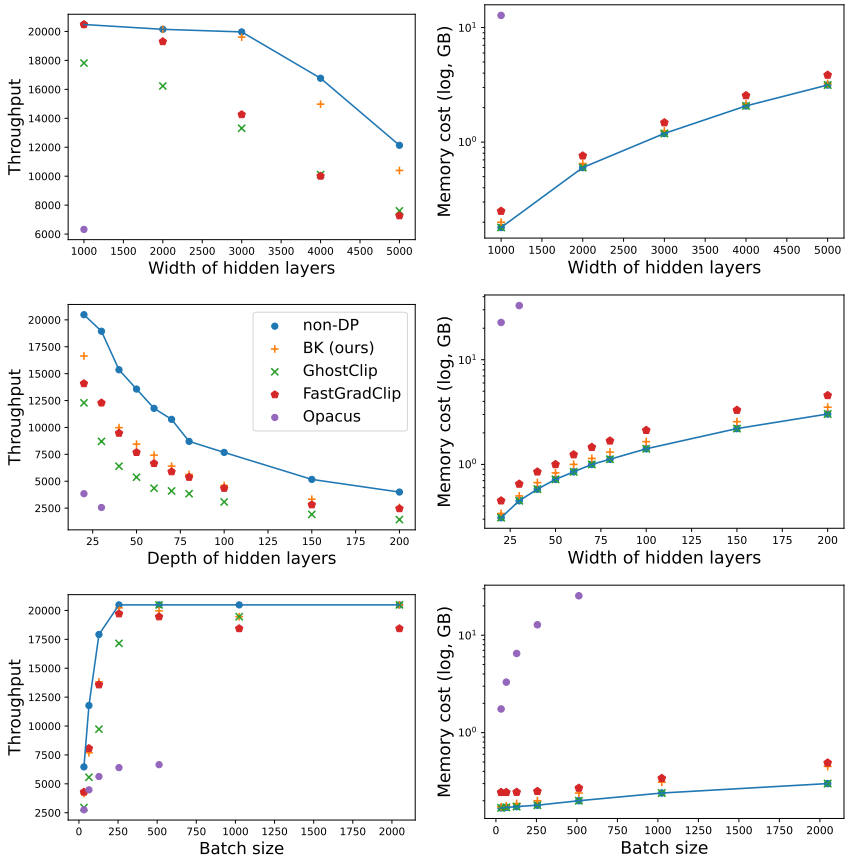

Figure 8: Ablation study of MLP on CIFAR10/CIFAR100 (images are flattened into vectors). Default model: 10 layers, width 1000, batch size 256.

	BK	Non-DP	GhostClip	Opacus
Time complexity	$6B\sum_l T_{(l)}p_{(l)}d_{(l)}$ $+\sum_l \left(\mathbb{I}\{2T_{(l)}^2 < p_{(l)}d_{(l)}\}\right.$ $\left.\cdot 2B\sum_l T_{(l)}^2(p_{(l)}+d_{(l)})\right)$	$6B\sum_l T_{(l)}p_{(l)}d_{(l)}$	$10B\sum_l T_{(l)}p_{(l)}d_{(l)}$ $+2B\sum_l T_{(l)}^2(p_{(l)}+d_{(l)})$	$8B\sum_l T_{(l)}p_{(l)}d_{(l)}$
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	$29.1*10^{12}$	$26.9*10^{12}(0.92\times)$	$47.1*10^{12}(1.61\times)$	$35.8*10^{12}(1.23\times)$
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	$6.4*10^9$	$6.1*10^9(0.95\times)$	$6.4*10^9(1.00\times)$	$28.6*10^9(4.46\times)$

Table 8: Time (upper half) and space (lower half) complexity of implementations ($B = 100$). For RoBERTa and GLUE datasets, $T = 256$; we use BK base ($\equiv$ BK-MixOpt). For vision transformers and ImageNet, $T = 224 \times 224$; we use BK-MixOpt. We mark the ratio of complexity to BK in ().

Model	Algorithm	Maximum batch size	Time/Epoch	Maximum throughput	Speedup by BK
RoBERTa-large SST-2	BK (ours)	41	13:33	83	—
	Non-private	51	9:50	114	0.73×
	GhostClip	48	17:34	64	1.30×
	Opacus	16	22:30	50	1.66×
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	2:22	296	—
	Non-private	157	1:47	393	0.75×
	GhostClip	156	2:54	242	1.22×
	Opacus	43	5:03	139	2.13×
GPT2-medium	BK (ours)	69	5:25	129	—
	Non-private	70	4:05	172	0.75×
	GhostClip	70	6:46	104	1.24×
	Opacus	15	14:22	49	2.63×
GPT2-large	BK (ours)	29	11:20	62	—
	Non-private	29	8:16	85	0.73×
	GhostClip	29	13:56	50	1.24×
	Opacus	5	44:05	16	3.88×
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) $\sum_l \min\{2T_{(l)}^2, p_{(l)}d_{(l)}\}$	Per-sample grad instantiation ($\sum_l p_{(l)}d_{(l)}$; # param)	Saving by MGN	Ghost norm ($\sum_l 2T_{(l)}^2 = 2H_{\text{out}}^2 W_{\text{out}}^2$)	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).

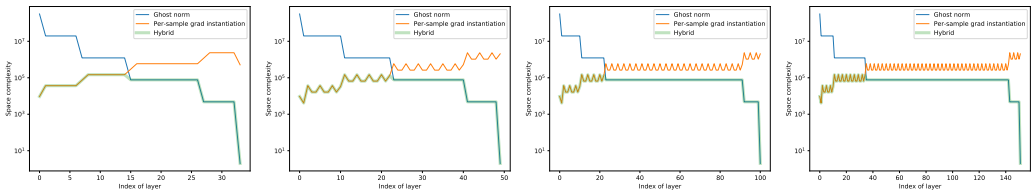

Figure 9: Layerwise space complexity of computing the per-sample gradient norm. Left to right: ResNet 34/50/101/152.

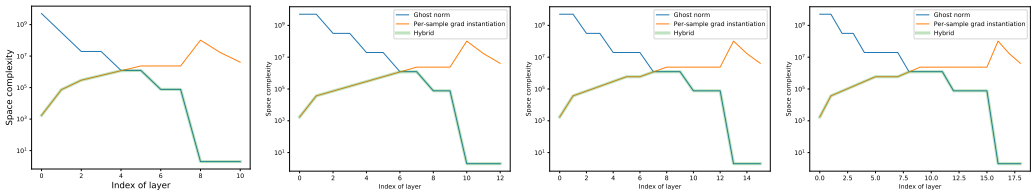

Figure 10: Layerwise space complexity of computing the per-sample gradient norm. Left to right: VGG 11/13/16/19.

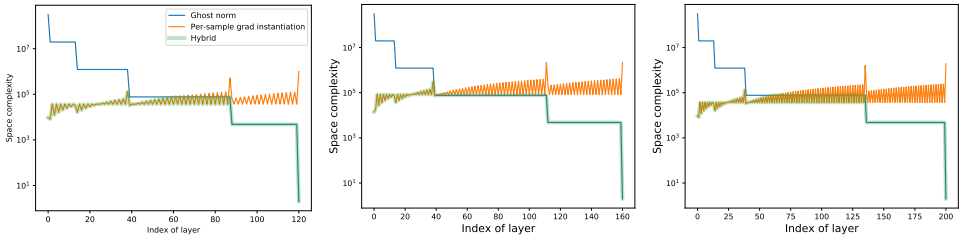

Figure 11: Layerwise space complexity of computing the per-sample gradient norm. Left to right: DenseNet 121/161/201.

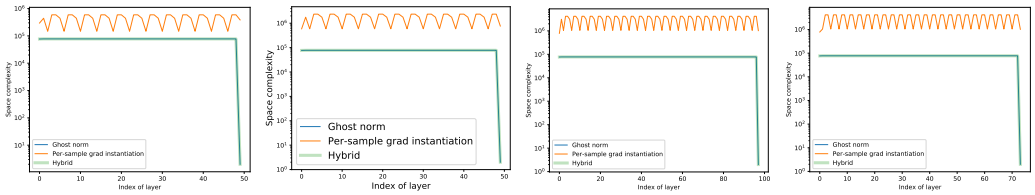

Figure 12: Layerwise space complexity of computing the per-sample gradient norm. Left to right: ViT small/base/large, and BEiT-large.

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.

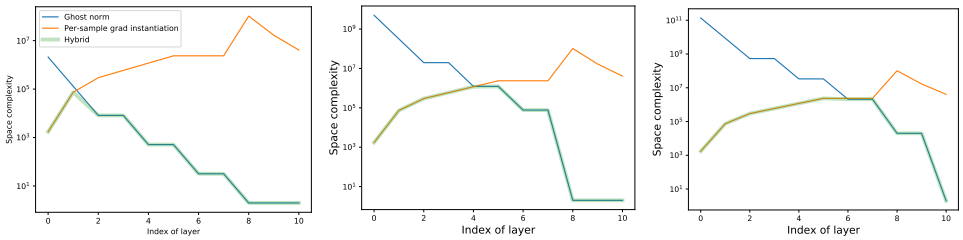

Figure 13: Layerwise space complexity of computing the per-sample gradient norm in VGG11.

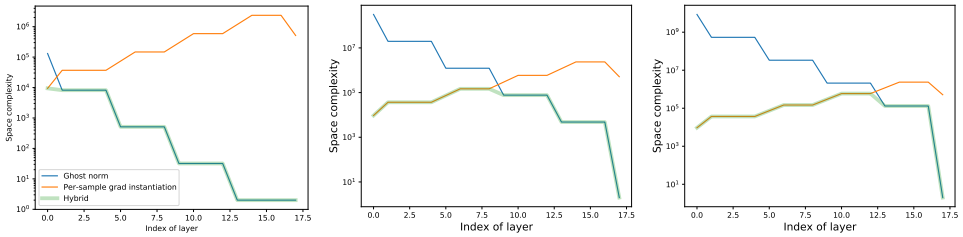

Figure 14: Layerwise space complexity of computing the per-sample gradient norm in ResNet18.

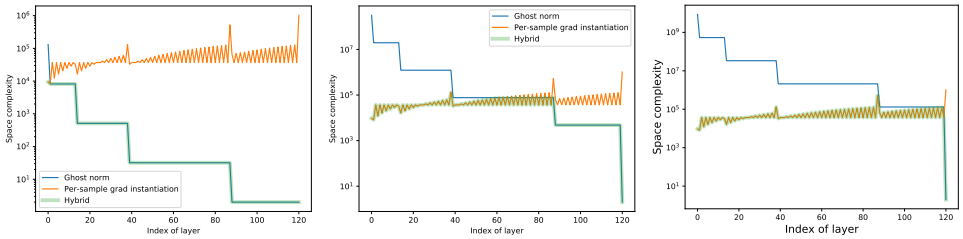

Figure 15: Layerwise space complexity of computing the per-sample gradient norm in DenseNet121.

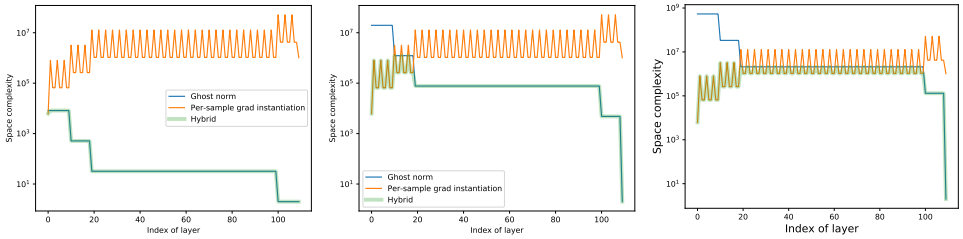

Figure 16: Layerwise space complexity of computing the per-sample gradient norm in ConvNeXT.

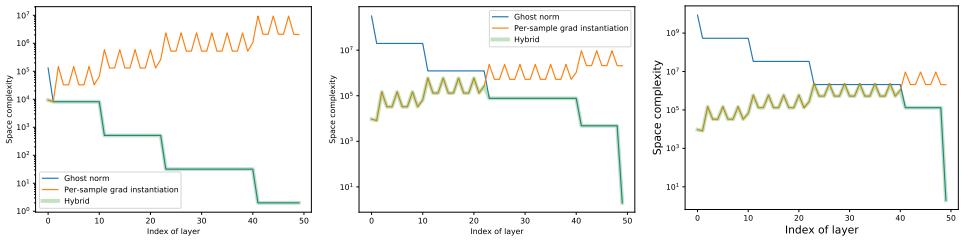

Figure 17: Layerwise space complexity of computing the per-sample gradient norm in Wide ResNet50.

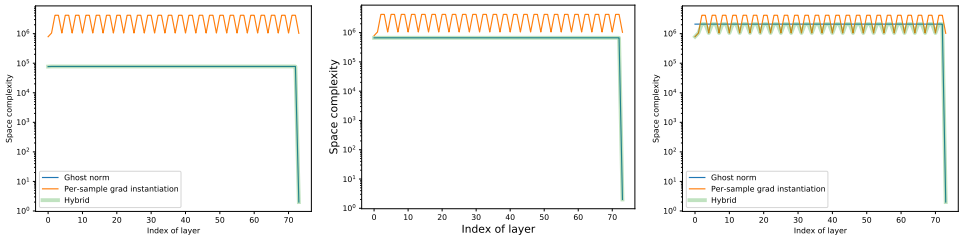

Figure 18: Layerwise space complexity of computing the per-sample gradient norm in BEiT-large.