Scaling Private Deep Learning with Opacus: Advances for Large Language Models

Sai Aparna Aketi [1] **Will Bullock** [1] **Iden Kalemaj** [1] **Enayat Ullah** [1] **Huanyu Zhang** [1]

Abstract

We introduce new features to Opacus, a free, open-source PyTorch library for training deep learning models with differential privacy. Opacus is designed for simplicity, flexibility, and speed. It provides a simple and user-friendly API, and enables researchers and machine learning practitioners to make a training pipeline private by adding as little as two lines to their code.

In this paper, we first provide a brief overview of Opacus, and then reveal the challenges posed by the prevalence of large language models (LLMs). To tackle these challenges, we introduce several recently added features including Fast Gradient and Ghost Clipping, model parallelism, parameter-efficient fine-tuning (PEFT), and mixed precision training.

1. Background and Introduction

Differential privacy (DP) (Dwork et al., 2006) has emerged as the leading notion of privacy for statistical analyses. It allows performing complex computations over large datasets while limiting disclosure of information about individual data points. Roughly stated, an algorithm that satisfies DP ensures that no individual sample in a database can have a significant impact on the output of the algorithm, quantified by the privacy parameters ε and δ.

Differentially Private Stochastic Gradient Descent (DP-SGD) was firstly introduced in Abadi et al. (2016). As a modification of the traditional SGD algorithm, it was designed to ensure differential privacy. Since its inception, DP-SGD has rapidly become the most widely adopted technique for training deep learning models while preserving data privacy.

[1]Central Applied Science, Meta, California, USA. Authors are listed in alphabetic order. Correspondence to: Huanyu Zhang <huanyuzhang@meta.com>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

Compared to traditional model updating methods such as SGD, DP-SGD differs in two crucial aspects:

- **Per-example gradient clipping:** We perform gradient clipping on each per-sample gradient such that the ℓ_2 norm is bounded, ensuring that individual gradients do not dominate the the size of the update.
- **Noise addition:** We aggregate clipped per-sample gradients over the mini-batch and add Gaussian noise to further protect the privacy of data.

However, for efficiency reasons, deep learning frameworks such as PyTorch or TensorFlow do not expose intermediate computations, including per-sample gradients; users only have access to the gradients averaged over a batch.

To address the challenges of training models with differential privacy, Yousefpour et al. (2021) introduced Opacus, an open-source library/framework built on top of PyTorch (with 1.8K stars and 3M downloads). Opacus is designed to provide a seamless experience for researchers and engineers, with three core principles in mind:

- **Simplicity:** Opacus offers a user-friendly API that allows users to train models with differential privacy without requiring in-depth knowledge of DP-SGD.
- **Flexibility:** Opacus supports most of the common PyTorch modules, which enables rapid prototyping by users proficient in PyTorch.
- **Speed:** Opacus seeks to minimize the performance overhead of DP-SGD by various optimizations and advanced features.

1.1. Example Usage

The primary interface for using Opacus is the `PrivacyEngine` class, which takes in the three core PyTorch training objects — model, optimizer, and data loader — and returns differentially private analogues of these objects. In fact, attaching Opacus to a non-private training flow can be done with just a few lines of code, as demonstrated in the following example:

```
data_loader = DataLoader()
model = Net()
optimizer = ADAM(model.parameters(), lr)

PrivacyEngine().make_private(
    module=model,
    optimizer=optimizer,
    data_loader=data_loader,
    ...,
    # Omitting less critical parameters
)
# Now it's business as usual
```

This simplicity makes it easy to adopt Opacus and start training differentially private models with minimal disruption to the existing workflows.

1.2. Challenges with DP-SGD for LLMs

As large language models (LLMs) become increasingly prevalent, ensuring the privacy of their training data has become a pressing issue. Research has shown that larger models are more prone to memorization, as demonstrated by studies such as Carlini et al. (2021).

One effective approach for addressing this challenge is training with DP-SGD. However, as model sizes continue to expand, supporting DP-SGD for these larger models poses significant infrastructure challenges:

- **Memory Constraints:** Naively implementing DP-SGD can lead to a substantial increase in memory cost, with the extra cost proportional to the batch size and model size. For example, with a FP32 model size of 100M and a batch size of 128, the extra memory cost required for DP-SGD is approximately 50 GB ($128 \times 100\text{M} \times 4$), which exceeds the memory capacity of an A100 GPU - 40GB.

  To address this issue, we introduced Fast Gradient and Ghost Clipping in version 1.5.0. By performing a second backward pass, we significantly reduced the extra memory requirements of DP-SGD, making them almost negligible compared to non-private settings. Further details on the technique can be found in Section 2.1.

- **Model Parallelism:** Although Distributed Data Parallel (DDP) has been supported in earlier versions of Opacus, larger models that cannot fit into a single GPU require model parallelism techniques such as Tensor Parallelism (TP) and Fully Sharded Data Parallel (FSDP). However, these techniques require additional work to support due to the unique challenges introduced by per-sample clipping. We discuss our efforts on FSDP in Section 2.2, which was released in June'25.

Method	Memory Overhead	Applicability
Opacus (previously)	$O(BLd^2)$	All layers
Fast Gradient Clipping	$O(Bd^2)$	All layers
Ghost Clipping	$O(BT^2)$	Linear-like layers

Table 1. A comparison of various techniques for per-sample clipping in DP-SGD. For simplicity, we analyze a network containing L identical linear layers, where d is the size of both input and output. B is the batch size , and T is the sequence length.

- **Throughput:** Due to the large size of the model and stringent computation requirements, it is essential to support common techniques leveraged in non-private model training to ensure efficient computation. We highlight our efforts on Parameter Efficient Fine-tuning (version 1.5.0) as well as Mixed Precision Training (released in June 2025). These techniques are discussed in Section 2.3 and Section 2.4, respectively.

2. Recent Developments and Initiatives

2.1. Fast Gradient and Ghost Clipping

The complexity introduced by DP-SGD is primarily the requirement for per-sample gradient clipping. Recall that native PyTorch only computes gradients averaged over a batch. Consider a simple example of an LLM with L identical layers stacked together, where both the input and output embedding sizes are d. For batch size B, a naive implementation of per-sample gradient clipping would either involve explicitly computing and instantiating the per-sample gradients in memory, which incurs a memory overhead of $O(BLd^2)$, or performing gradient accumulation, which results in $O(B)$ sequential gradient computations. Previously, Opacus used the former method, which restricted its applicability to larger models.

We introduce Fast Gradient Clipping techniques (Lee & Kifer, 2021; Bu et al., 2022) to Opacus (Ullah et al., 2024). The key idea behind these techniques is based on the following observation: suppose per-sample gradient norms are known, then gradient clipping can be achieved by backpropagation on a re-weighted loss function $\tilde{\ell}$, defined as $\tilde{\ell} = \sum_i R_i \ell_i$, where ℓ_i are per-sample losses, and $R_i = min(\frac{C}{C_i}, 1)$ are the clipping coefficients computed from the per-sample gradient norms C_i and clipping upper bound C.

The above idea may seem circular at first glance, as it appears to require instantiating per-sample gradients in order to calculate per-sample gradient norms. However, for widely-used components of neural network architectures, such as linear layers, it is indeed possible to obtain per-sample gradient norms in a single backward pass without the need for instantiating per-sample gradients. This sug-

gests a workflow that involves two backward passes: the first to compute per-sample gradient norms and re-weighted loss $\tilde{\ell}$, and the second to compute the aggregated (not per-sample) clipped gradient.

Fast Gradient Clipping (FGC) (Lee & Kifer, 2021). In FGC, the per-sample gradient norm is calculated one layer at a time: *(a).* For each layer, the per-sample gradient is instantiated and its norm is calculated. *(b).* The per-sample gradient is then immediately discarded. *(c).* The (squared) per-sample gradient norms of each layer are summed up to obtain the overall (squared) per-sample gradient norm. By performing this operation one layer at a time, FGC reduces the memory overhead from $O(BLd^2)$ to $O(Bd^2)$.

Ghost Clipping (GC) (Li et al., 2021). Extending the above, Ghost Clipping uses the fact that for linear layers and their generalizations, per-sample gradient norms can be calculated directly from the activation gradients, `backprops`, and `activations`, of size $B \times T \times d$ each. The per-sample gradients are the outer product of the two, taking $O(BTd^2)$ time and $O(Bd^2)$ space. Ghost clipping instead calculates the (squared) norm of the gradient as the sample-wise product of the (squared) norm of `backprops` and `activations`. This takes $O(BTd^2)$ time and $O(BT^2)$ space. If the sequence length T is much smaller than the embedding length d, GC uses even less memory than FGC.

Please refer to Table 1 for a comparison of the three techniques. As an example, for the task of privately fine-tuning last three layers (100M parameters) of BERT for a text classification task on a 16GB P100, we observe that the earlier version of Opacus OOMs on a batch size of 512, where as Ghost Clipping supports a batch size of upto 1024, same as that by (non-private) PyTorch.

2.2. Model Parallelisms

As the demand for private training of large-scale models, such as Large Language Models (LLMs), continues to grow, it is crucial for Opacus to support both data and model parallelism techniques.

Previously, Opacus only supported Differentially Private Distributed Data Parallel (DP-DDP) to enable large-scale multi-GPU training. While DP-DDP effectively scales model training across multiple GPUs and nodes, it requires each GPU to store a copy of the model and optimizer states, leading to high memory requirements, especially for large models. This limitation underscores the need for alternative parallelization techniques, such as Fully Sharded Data Parallel (FSDP), which can offer improved memory efficiency and increased scalability (Zhao et al., 2023).

In the context of training large language models, different parallelism strategies are typically employed based on model size:

- 1D Parallelism: DDP or Fully Sharded Data Parallel (FSDP) for small-sized models (< 10 billion parameters).
- 2D Parallelism: FSDP combined with Tensor Parallelism (TP) for medium-sized models ($10 - 100$ billion parameters).
- 4D Parallelism: FSDP combined with TP, Pipeline Parallelism (PP), and Context Parallelism (CP) for large-sized models (> 100 billion parameters).

Enabling FSDP is the first step towards achieving 2D and 4D parallelism with Opacus, paving the way for more efficient and scalable private training or fine-tuning of medium to large-scale models.

Setup	Parallelism	Batch-size	Samples per second
AdamW (7.5B)	DP-DDP	8	OOM
AdamW (7.5B)	FSDP	64	**12.58** $\pm$ 0.13
SGD (5.1B)	DP-DDP	64	9.30 $\pm$ 0.43
SGD (5.1B)	FSDP	64	**15.37** $\pm$ 0.12

Table 2. Full fine-tuning of Llama-3 8B on a synthetic dataset, maximum sequence length of 512, 1x8 A100 80GB GPUs.

We experiment with full fine-tuning of the Llama-3 8B model on a synthetic dataset. Table 2 presents the throughput in terms of samples/inputs processed per second. Currently, FSDP with Ghost Clipping doesn't support tied parameters (embedding layers). We freeze these layers during fine-tuning which brings the trainable parameters down from $8B \rightarrow 7.5B$. As shown in Table 3, DP-DDP throws OOM error even with a batch size of one per device. With FSDP, each device can fit a batch size of 8, enabling full fine-tuning of Llama-3 8B.

To compare full fine-tuning of FSDP with DP-DDP, we shift from AdamW optimizer to SGD w/o momentum and reduce the trainable parameters from $7.5B \rightarrow 5.1B$ by freezing normalization layers' and gate projection layers' weights. This allows DP-DDP to run with a batch-size of 2. In this setting, we observe that FSDP is $1.65\times$ times faster than DP-DDP for the same batch-size.

The introduction of FSDP in Opacus (Opacus, 2025b) marks a significant advancement, offering a scalable and memory-efficient solution for private training of LLMs. This development not only enhances the capability of Opacus to handle large-scale models but also sets the stage for future integration of other model parallelism strategies. Looking ahead, our focus will be on enabling Tensor Parallelism and 2D parallelism for Opacus.

2.3. Parameter-Efficient Fine Tuning

Parameter-efficient fine-tuning (PEFT) has become popular for reducing the number of trainable parameters, which

leads to decreased memory usage and enhanced computational efficiency. The noise introduced by DP-SGD increases with the number of trainable model parameters, thus creating a larger utility gap between non-private and private training as the model size grows. Consequently, DP-SGD has been proved to be most effective in transfer learning scenarios, where a well-performing model trained on public data is fine-tuned with DP-SGD on a private dataset.

Techniques such as low-rank adaptation (LoRA) (Hu et al., 2022) are particularly noteworthy, as they fine-tune only a small number of additional model parameters, thereby reducing the number of trainable parameters with minimal impact on utility. For DP-SGD training, these techniques are essential to avoid the curse of dimensionality caused by noise, as well as to improve training efficiency.

We publish a tutorial (Opacus, 2025a) showcasing that DP-SGD is compatible and can be used together with the `peft` library from HuggingFace (HuggingFace, 2025) for parameter-efficient fine-tuning. This is achieved with no conceptual changes to the privacy analysis. For the task of fine-tuning the last few layers of a BERT model on a common NLP dataset, we show that LoRA fine-tuning with DP-SGD achieves on par test set accuracy (74.0%) compared to normal DP-SGD (74.3%), while training 100x fewer parameters.

2.4. Mixed Precision Training

Mixed precision is the combined use of numerical precision in a workload. Half-precision (e.g., BF16) operations compared to single-precision (FP32) operations enable training larger models, with larger batch sizes, faster operations and faster data transfers. Mixed precision training has been successfully used in speeding up training of LLMs and reducing memory utilization, while maintaining on-par utility compared to full precision (Micikevicius et al., 2018).

At a high level, mixed precision training uses BF16 for the forward and backward pass and FP32 for weight updates. Full precision weight updates are necessary for maintaining utility, as weight updates can at times become extremely small or large and full precision helps maintain stable operations (Peng et al., 2023). Some layers, such as normalization layers, will also perform operations in FP32 to maintain numerical stability.

PyTorch supports mixed precision training through the `torch.amp` package (PyTorch, 2025). We enable support for mixed precision training with Opacus via `torch.amp`, which was previously unavailable. We also support low precision training (e.g., BF16 only). Using either setting requires only a few extra lines of code from the user side, which are the same as for non-private training.

We experiment with fine-tuning a pre-trained BERT-base model: we fine-tune either the last few layers (8M out of 100M trainable parameters) or fine-tune all layers with LoRA (500k parameters). We use either FP32 only, BF16 only, or mixed precision. Hyperparameters are the same across all settings. In Table. 3, we compare the difference in peak memory and time taken (averaged over 10 steps). BF16-only training achieves 2x memory improvement and 2x speed-up compared to FP32. Mixed precision training achieves similar gains in speed but smaller memory improvements, due to the storage of full-precision weights.

Table 3. Memory, time, and utility for different precision settings when fine-tuning BERT with Opacus (batch size = 32).

Fine-tuning setup	**Precision setting**	Peak memory improvement over baseline	Time per iteration improvement over baseline	Test set accuracy after 1 epoch ($\varepsilon = 2$)
Fine-tune last few layers	High Precision (Baseline)	1.00×	1.00×	0.7287
	Mixed Precision	1.41×	2.00×	0.7270
	Low Precision	1.99×	2.00×	0.6827
Fine-tune all layers with LoRA	High Precision (Baseline)	1.00×	1.00×	0.7193
	Mixed Precision	1.16×	1.64×	0.7207
	Low Precision	1.87×	2.00×	0.7250

In Table 3, we also compare the test-set accuracy after 1 training epoch for the different precision settings. When fine-tuning the last few layers, mixed precision and FP32 training achieve on par performance, while low precision training incurs a significant decrease in utility.

With LoRA fine-tuning, the highest accuracy is achieved with BF16, while mixed and high precision training are on-par. We hypothesize that low precision training with DP-SGD performs best when fine-tuning only linear layers, as in LoRA, but harms utility when other types of layers are involved, such as normalization layers, which normally require high precision operations.

Mixed precision or low precision training with Opacus enables training of larger models with larger batch sizes in memory-constrained settings. Recent foundation models train at even lower precision such as FP8 (Meta, 2025). Further research is required to understand the utility performance of DP-SGD at more reduced precision settings.

3. Conclusion

In this paper, we have introduced significant advancements in Opacus, addressing the growing demand for privacy-preserving techniques, especially with LLMs. Key features such as Fast Gradient and Ghost Clipping, model parallelism with FSDP, and mixed precision training enhance the efficiency and scalability of training large models while maintaining privacy guarantees. Looking ahead, we plan to incorporate additional parallelism strategies, like Tensor Parallelism. These efforts ensure that Opacus remains at the forefront of enabling privacy-preserving machine learning for the research community.

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.

Bu, Z., Mao, J., and Xu, S. Scalable and efficient training of large convolutional neural networks with differential privacy. In *NeurIPS*, 2022.

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.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pp. 265–284. Springer, 2006.

Hu, J. E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., and Chen, W. LoRA: Low-rank adaptation of large language models. In *ICLR*, 2022.

HuggingFace. Peft: Parameter-efficient fine-tuning. https://huggingface.co/docs/peft/en/index, 2025. Accessed: 2025-05-24.

Lee, J. and Kifer, D. Scaling up differentially private deep learning with fast per-example gradient clipping. In *PETS*, 2021.

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

Meta. Llama-4-maverick-17b-128e-instructions. https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct, 2025. Accessed: 2025-05-24.

Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. In *ICLR*, 2018.

Opacus. Building a text classifier with differential privacy. https://opacus.ai/tutorials/building_text_classifier, 2025a. Accessed: 2025-05-24.

Opacus. Enabling fully sharded data parallel (fsdp) in opacus. https://pytorch.org/blog/enabling-fully-sharded-data-parallel-fsdp2-in-opacus/, 2025b. Accessed: 2025-07-07.

Peng, H., Wu, K., Wei, Y., Zhao, G., Yang, Y., Liu, Z., Xiong, Y., Yang, Z., Ni, B., Hu, J., Li, R., Zhang, M., Li, C., Ning, J., Wang, R., Zhang, Z., Liu, S., Chau, J., Hu, H., and Cheng, P. FP8-LM: Training FP8 large language models, 2023. URL https://arxiv.org/abs/2310.18313.

PyTorch. Automatic mixed precision (amp) package. https://docs.pytorch.org/docs/stable/amp.html, 2025. Accessed: 2025-05-24.

Ullah, E., Zhang, H., Bullock, W., and Mironov, I. Enabling fast gradient clipping and ghost clipping in Opacus. https://pytorch.org/blog/clipping-in-opacus/, 2024. Accessed: 2025-05-25.

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

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. PyTorch FSDP: Experiences on scaling fully sharded data parallel. In *PVLDB*, 2023.