Opacus: User-Friendly Differential Privacy Library in PyTorch

Ashkan Yousefpour[*] **Igor Shilov**[*] **Alex Sablayrolles**[*] **Davide Testuggine**

Karthik Prasad **Mani Malek** **John Nguyen** **Sayan Ghosh**

Akash Bharadwaj **Jessica Zhao**[*] **Graham Cormode** **Ilya Mironov**

Meta AI

Abstract

We introduce Opacus, a free, open-source PyTorch library for training deep learning models with differential privacy (hosted at `opacus.ai`). Opacus is designed for simplicity, flexibility, and speed. It provides a simple and user-friendly API, and enables machine learning practitioners to make a training pipeline private by adding as little as two lines to their code. It supports a wide variety of layers, including multi-head attention, convolution, LSTM, GRU (and generic RNN), and embedding, right out of the box and provides the means for supporting other user-defined layers. Opacus computes batched per-sample gradients, providing higher efficiency compared to the traditional "micro batch" approach. In this paper we present Opacus, detail the principles that drove its implementation and unique features, and benchmark it against other frameworks for training models with differential privacy as well as standard PyTorch.

1 Background and Introduction

Differential privacy (DP) [5] 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.

Formally, a randomized mechanism $M\colon \mathcal{D} \to \mathcal{R}$ is (ϵ, δ)-differentially private for $\epsilon > 0$ and $\delta \in [0, 1)$ if for any two neighboring datasets $D, D' \in \mathcal{D}$ (i.e., datasets that differ in at most one sample) and for *any* subset of outputs $R \subseteq \mathcal{R}$ it holds that

$$\mathbb{P}(M(D) \in R) \leq \exp(\epsilon)\, \mathbb{P}(M(D') \in R) + \delta.$$

Differentially Private Stochastic Gradient Descent (DP-SGD) due to Abadi et al. [1], building on Song et al. [18] and Bassily et al. [2], is a modification of SGD that ensures differential privacy on every model parameters update. Instead of computing the average of gradients over a batch of samples, a DP-SGD implementation computes per-sample gradients, clips their ℓ_2 norm, aggregates them into a batch gradient, and adds Gaussian noise (see Fig. 1 for an illustration.) However, mainly for efficiency reasons, deep learning frameworks such as PyTorch or TensorFlow do not expose

[*]Equal contribution. Jessica Zhao contributed benchmarking code and analysis (Section 3). Correspondence to `yousefpour@fb.com`.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

intermediate computations, including per-sample gradients; users only have access to the gradients averaged over a batch.

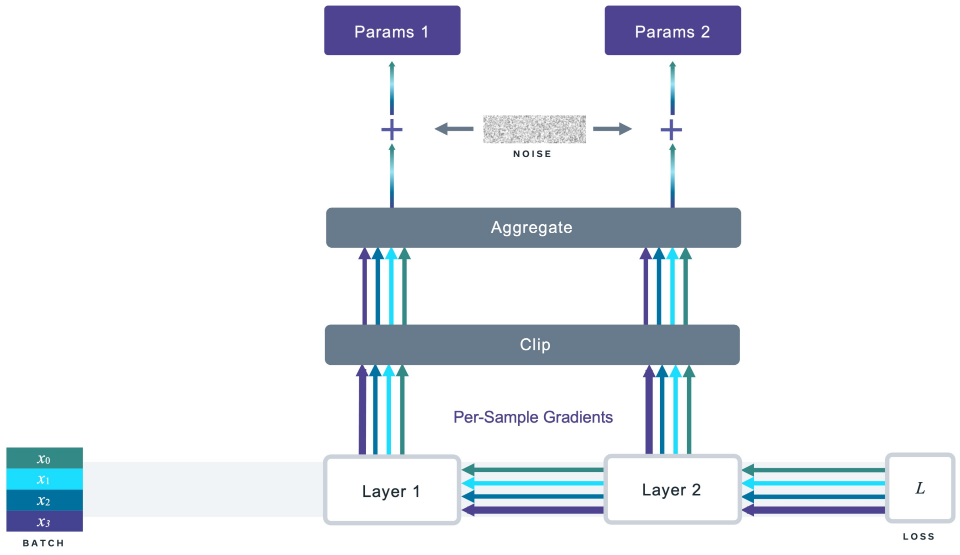


Figure 1: Pictorial representation of the DP-SGD algorithm. The single-colored lines represent per-sample gradients, the width of the lines represent their respective norms, and the multi-colored lines represent the aggregated gradients.

A naïve way to implement DP-SGD is thus to separate each batch into micro-batches of size one, compute the gradients on these micro-batches, clip, and add noise (see Appendix A for sample code to obtain the per-sample gradients using this approach). While this procedure (called the "micro-batch method" or "micro-batching") does indeed yield correct per-sample gradients, it can be very slow in practice due to underutilization of hardware accelerators (GPUs and TPUs) that are optimized for batched, data-parallel computations.

Opacus implements performance-improving vectorized computation instead of micro-batching. In addition to speed, Opacus is designed to offer simplicity and flexibility. In this paper, we discuss these design principles, highlight some unique features of Opacus, and evaluate its performance in comparison with other DP-SGD frameworks.

2 Design Principles and Features

Opacus is designed with the following three principles in mind:

- *Simplicity*: Opacus exposes a compact API that is easy to use out of the box for researchers and engineers. Users need not know the details of DP-SGD in order to train their models with differential privacy.
- *Flexibility*: Opacus supports rapid prototyping by users proficient in PyTorch and Python, thanks to its rich set of features (described below).
- *Speed*: Opacus seeks to minimize performance overhead of DP-SGD by supporting vectorized computation.

We explain throughout the paper how these principles manifest themselves in the Opacus API.

Example Usage. The main entry point to Opacus is the `PrivacyEngine` class. It keeps track of "privacy budget" spent so far and is responsible for wrapping regular PyTorch training objects with DP-related code. The key method provided by `PrivacyEngine` is `make_private()`. It takes the three PyTorch training objects—model, optimizer and data loader— along with the privacy parameters (noise multiplier and maximum norm of the gradients) and outputs differentially private analogues of these objects:

- the model wrapped with `GradSampleModule`, which adds the ability to compute per-sample gradients;
- the optimizer wrapped with an additional code for clipping gradients and adding noise;
- the data loader transformed into one using Poisson sampling as required by DP-SGD.

This design provides a good balance between *simplicity* and *flexibility*. On the one hand, most users are able to switch to DP training by calling a single method. On the other hand, advanced users can modify the details of the private components' behavior as long as their interface is unchanged.

Attaching Opacus to an existing script can be done with changing as few as two lines of code, e.g., the lines containing `privacy_engine` in the following example:

```
dataset = Dataset()
model = Net()
optimizer = SGD(model.parameters(), lr)
data_loader = torch.utils.data.DataLoader(dataset, batch_size)
privacy_engine = PrivacyEngine()
model, optimizer, data_loader = privacy_engine.make_private(
   module=model,
   optimizer=optimizer,
   data_loader=data_loader,
   noise_multiplier=noise_multiplier,
   max_grad_norm=max_grad_norm,
)
# Now it's business as usual
```

Main features of Opacus. We highlight some of the key features Opacus provides.

Privacy accounting. Opacus provides out of the box privacy tracking with an accountant based on Rényi Differential Privacy [13, 14]. The `PrivacyEngine` object keeps track of how much privacy budget has been spent at any given point in time, enabling early stopping and real-time monitoring. Opacus also allows a user to directly instantiate a DP training targeting an (ϵ, δ) budget. In this instance, the engine computes a noise level σ that yields an overall privacy budget of (ϵ, δ). Opacus also exposes an interface to write custom privacy accountants.

Model validation. Before training, Opacus validates that the model is compatible with DP-SGD. For example, certain layers (e.g., `BatchNorm` or `GroupNorm` modules in some configurations) mix information across samples of a batch, making it impossible to define a per-sample gradient; Opacus disallows those modules. It also ensures that no additional statistics without DP guarantees are tracked by the model (See Appendix C).

Poisson sampling. Opacus also supports uniform sampling of batches (also called Poisson sampling): each data point is independently added to the batch with probability equal to the sampling rate. Poisson sampling is necessary in some analyses of DP-SGD [14].

Efficiency. Opacus makes efficient use of hardware accelerators (See Appendix B). Opacus also supports distributed training via PyTorch's `DistributedDataParallel`.

Virtual steps. As Opacus is highly optimized for batched per-sample gradient computation, it faces an inevitable speed/memory trade-off. In particular, when computing per-sample gradients for the entire batch, the size of the gradient tensor it needs to store is increased by the factor of `batch_size`. To support a wider range of batch sizes, Opacus provides an option to decouple physical batch sizes, limited by the amount of memory available, and logical batch sizes, whose selection is driven by considerations of model convergence and privacy analysis.

Predefined and custom layers. Opacus comes with several predefined layer types, including convolution, multi-head attention, LSTM, GRU (and generic RNN), normalization, and embedding layers. Moreover, it allows users to add their own custom layers. When adding a custom layer, users can provide a method to calculate per-sample gradients for that layer and register it with a simple decorator provided by Opacus. The details can be found in `opacus.ai/tutorials`.

Secure random number generation. Opacus offers a cryptographically safe (but slower) pseudo-random number generator (CSPRNG) for security-critical code. This can be enabled by the option `secure_mode`, which enables CSPRNG for noise generation and random batch composition.

Noise scheduler and variable batch size. Similar to learning rate scheduler in deep learning, the noise scheduler in Opacus adjusts the noise multiplier during training by evolving it according to some predefined schedule, such as exponential, step, and custom function. Opacus also supports varying batch sizes throughout training.

Modular. Opacus integrates well with PyTorch Lightning, a high-level abstraction framework for PyTorch, which reduces boilerplate and simplifies coding complex networks.

3 Benchmarks

We benchmark Opacus against other frameworks for training models with DP-SGD as well as standard PyTorch by comparing their respective runtimes on several end-to-end model training tasks. We also quantify the runtime and memory overhead of training with DP using Opacus compared to training without DP using PyTorch for each layer that Opacus currently supports.

3.1 End-to-end benchmarks

Our end-to-end benchmarks are based on the Fast-DPSGD benchmarks [19]. We evaluate Opacus on four end-to-end model training tasks against a JAX implementation of DP-SGD, a custom TensorFlow Privacy implementation, BackPACK, and PyVacy, as well as standard PyTorch without DP.

3.1.1 Frameworks

JAX is a general-purpose framework for high-performance numerical computing that uses just-in-time (JIT) compilation and static graph optimization. We use the custom implementation of DP-SGD found in [19] and denote it as *JAX (DP)*.

TensorFlow Privacy is a TensorFlow library for differentially private model training. We use the custom implementation with vectorization and XLA-driven JIT compilation, which outperforms both the custom TensorFlow Privacy implementation without XLA and the existing TensorFlow Privacy library in [19]. We denote it as *Custom TFP (XLA)* for consistency with [19].

To enable per-sample gradient extraction, BackPACK extends several PyTorch layers with support for efficient Jacobian computation. In contrast, PyVacy processes each sample individually in a for-loop, forgoing parallelization.

3.1.2 Experimental setup

Following [19], we train a CNN with 26,010 parameters on MNIST [9], a handwritten digit recognition dataset, and a CNN with 605,226 parameters on CIFAR-10 [8], a dataset of small color images. We train an embedding network and an LSTM network with 160,098 and 1,081,002 parameters respectively on the IMDb dataset [12], which consists of movie reviews for sentiment classification.

For each model and framework, we train the model using the framework's implementation of DP-SGD with a given privacy budget. Since Opacus is built on top of PyTorch, we also train each model using PyTorch *without* DP to better understand the runtime overhead of enabling DP with Opacus compared to training without DP using PyTorch.

We benchmark the latest version of each framework as of December 8th, 2021 in a separate Docker container, see Appendix E.3 for details. Compared to the setup in [19], our GPU has more VRAM (40GB rather than 12GB), which allows us to benchmark larger batch sizes (512, 1024, 2048) for a more extensive comparison. The code is available at `https://github.com/TheSalon/fast-dpsgd/tree/latest`.

We report each framework's median per-epoch runtime on each end-to-end training task at various batch sizes in Table 1.

3.1.3 Results

JAX (DP) consistently achieves the lowest runtime among all DP-SGD frameworks on both MNIST and IMDb with the embedding network, even outperforming PyTorch without DP at smaller batch sizes. On CIFAR-10, JAX (DP) outperforms all other frameworks at smaller batch sizes, while

Batch Size	16	32	64	128	256	512	1024	2048
JAX (DP)	3.72	1.93	0.94	0.52	0.26	0.18	0.16	0.15
PyTorch without DP	*5.82*	*2.97*	*1.55*	*0.82*	*0.47*	*0.26*	*0.16*	*0.11*
Opacus	15.81	8.00	4.25	2.30	1.22	0.64	0.36	0.21
BackPACK	21.41	11.00	5.62	3.14	1.83	1.31	1.41	1.31
Custom TFP (XLA)	13.55	10.61	9.05	8.30	7.87	7.56	7.32	7.44
PyVacy	109.08	106.31	107.96	108.47	109.13	110.94	110.15	112.28

(a) MNIST with CNN

Batch Size	16	32	64	128	256	512	1024	2048
JAX (DP)	10.20	5.98	3.64	3.28	2.91	2.75	2.66	2.60
PyTorch without DP	*11.07*	*5.99*	*3.42*	*1.76*	*1.07*	*0.87*	*0.82*	*0.79*
Opacus	32.02	16.59	8.66	4.40	2.45	2.06	1.93	1.89
BackPACK	46.57	24.17	13.09	10.66	10.91	10.36	10.06	10.02
Custom TFP (XLA)	41.51	37.45	33.66	33.07	32.18	30.78	31.82	30.73
PyVacy	198.35	196.63	192.54	191.75	192.27	195.36	192.90	192.76

(b) CIFAR-10 with CNN

Batch Size	16	32	64	128	256	512	1024	2048
JAX (DP)	0.74	0.40	0.19	0.10	0.06	0.05	0.05	0.04
PyTorch without DP	*1.54*	*0.78*	*0.40*	*0.21*	*0.11*	*0.06*	*0.04*	*0.04*
Opacus	3.49	1.76	0.88	0.45	0.23	0.12	0.11	0.12
Custom TFP (XLA)	1.93	1.11	0.68	0.47	0.37	0.31	0.29	0.29
PyVacy	18.73	18.71	18.43	18.58	18.63	18.93	18.00	18.65

(c) IMDb with Embedding

Batch Size	16	32	64	128	256	512	1024	2048
JAX (DP)	25.61	12.89	8.68	6.48	5.23	4.54	4.33	4.22
PyTorch without DP	*14.08*	*7.13*	*3.91*	*2.00*	*1.04*	*0.60*	*0.41*	*0.34*
Opacus	358.80	183.22	118.11	59.92	30.45	15.91	11.51	9.94
Custom TFP (XLA)	21.99	12.30	8.80	5.03	3.08	2.21	1.65	1.44
PyVacy	201.27	201.67	200.49	200.36	200.48	201.55	200.63	202.34

(d) IMDb with LSTM

Table 1: Median runtime (in seconds) per epoch, computed over 20 epochs, of several DP-SGD frameworks as well as PyTorch without DP on four end-to-end model training tasks at various batch sizes. Since BackPACK does not support embedding or LSTM layers, the corresponding rows are omitted.

Opacus surpasses JAX (DP) at larger batch sizes. On IMDb with the LSTM network, Custom TFP (XLA) consistently achieves the lowest runtime among all DP-SGD frameworks. Training with PyVacy consistently results in the highest runtime due to its use of micro-batching.

At a batch size of 2048, Opacus achieves the lowest runtime on CIFAR-10 and the second lowest runtime after JAX (DP) on MNIST and IMDb with the embedding network, its runtime being $1.4\times$ and $3\times$ that of JAX (DP) respectively. On IMDb with the LSTM network, Opacus achieves the third lowest runtime after Custom TFP (XLA) and JAX (DP), with $7\times$ the runtime of Custom TFP (XLA) and $2.4\times$ the runtime of JAX (DP).

While the median per-epoch runtime decreases as the batch size increases for most frameworks, the effect is strongest for Opacus and PyTorch: By increasing the batch size from 16 to 2048, Opacus's per-epoch runtime decreases by a factor ranging from $17\times$ (on CIFAR-10) to $75\times$ (on MNIST). We report the mean per-epoch runtime reduction from increasing the batch size from 16 to 2048 for each framework: $40\times$ (Opacus), $37.8\times$ (PyTorch without DP), $12.8\times$ (JAX (DP)), $10.5\times$ (BackPACK), $6.3\times$ (Custom TFP (XLA)), and $1\times$ (PyVacy).

Since Opacus and PyTorch benefit the most from a larger batch size, increasing the batch size further may close the gap (where applicable) between Opacus and JAX (DP) or Custom TFP (XLA) and even result in Opacus outperforming them. Both JAX (DP) and Custom TFP (XLA) rely on JIT compilation, which incurs a large runtime overhead in the first epoch (up to $101\times$ and $625\times$ the median per-epoch runtime, respectively) in exchange for a lower runtime in subsequent epochs. See Fig. 4 for each framework's cumulative runtime over 20 epochs.

On MNIST, CIFAR-10, and IMDb with the embedding network, enabling DP with Opacus incurs a $2\times$ to $2.9\times$ runtime overhead compared to training without DP using PyTorch. On IMDb with the LSTM network, the runtime overhead ranges from $25\times$ to $30\times$, see Section 3.2.3 for further analysis. Since Opacus is built on PyTorch, we expect any future improvements to PyTorch's efficiency (e.g. `torch.vmap` graduating from the prototype stage) to benefit Opacus as well.

3.2 Microbenchmarks

We measure the runtime and peak allocated memory for one forward and one backward pass for each layer currently supported by Opacus, both with and without DP. We report the runtime and memory overhead of enabling DP with Opacus for each layer.

3.2.1 Experimental setup

For each layer that Opacus currently supports, we benchmark both the layer with DP enabled and the corresponding `torch.nn` module without DP at various batch sizes.

For the convolutional, normalization, linear, and embedding layers, which Opacus supports directly, wrapping the corresponding `torch.nn` module in Opacus's `GradSampleModule` enables DP. For the multi-head attention and RNN-based layers, which Opacus provides custom implementations for, wrapping the corresponding custom module in `GradSampleModule` enables DP.

Each benchmark estimates the mean runtime and peak allocated CUDA memory for one forward and one backward pass by measuring the cumulative runtime and peak allocated CUDA memory for a total of 2,000 forward and backward passes on 100 different input batches. See Appendix E.3 for details. The microbenchmarking code is available at `https://github.com/pytorch/opacus/tree/main/benchmarks`.

We report the runtime and peak memory overhead of the DP-enabled layer relative to the corresponding `torch.nn` module for all currently supported layers in Fig. 2.

3.2.2 Memory requirements of DP

Since DP-SGD requires per-sample gradients, training with DP-SGD uses more memory than training without DP-SGD. Let L denote the number of trainable parameters in a given module and let each parameter be of size 1. Let C be the size of the features, label, and model output for a single data point. Let M denote the total memory usage for one forward and one backward pass on a batch of size b. Then, ignoring intermediate computations and constant additive overhead such as from

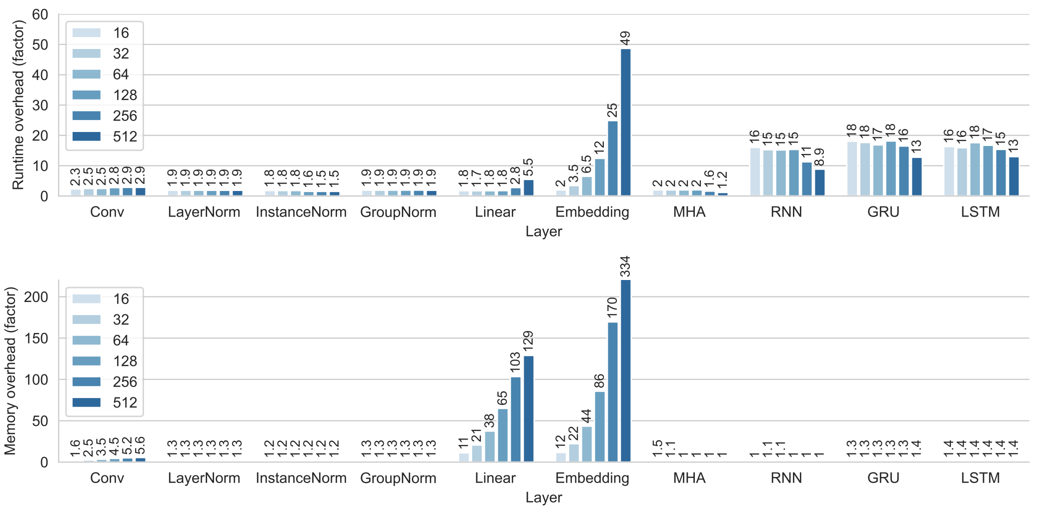


Figure 2: Runtime and peak allocated memory overhead of enabling DP for each layer currently supported by Opacus at various batch sizes. Top: Runtime overhead (factor). Bottom: Peak allocated memory overhead (factor). The runtime overhead is the mean runtime for one forward and one backward pass of the DP-enabled layer divided by the mean runtime for one forward and one backward pass of the corresponding `torch.nn` module without DP. The overhead in terms of peak allocated memory is calculated in the same manner.

non-trainable parameters:

$$M_{\text{non-DP}} = bC + 2L, \tag{1}$$
$$M_{\text{DP}} = bC + (1+b)L. \tag{2}$$

In both non-DP and DP training, the features, labels, and the module's output for b data points occupy memory of size bC, and the module itself occupies memory of size L by definition. Without DP, we expect the gradient to occupy memory of size L as well, whereas with DP, we expect the gradient to occupy memory of size bL due to b per-sample gradients. For $b \gg 1$, we can approximate the memory overhead as follows:[†]

$$\frac{M_{\text{DP}}}{M_{\text{non-DP}}} \approx \begin{cases} \frac{bC+(1+b)L}{bC} \approx 1 + \frac{L}{C} & \text{if } L/C \ll b \\ \frac{2+b}{3} \approx \frac{b}{3} & \text{if } L/C \approx b \\ \frac{1+b}{2} \approx \frac{b}{2} & \text{if } L/C \gg b \end{cases} \tag{3}$$

3.2.3 Results

Runtime. For the convolutional, normalization, and multi-head attention layers, enabling DP with Opacus's `GradSampleModule` results in a 1.2× to 2.9× runtime increase, which we attribute to the calculation of per-sample gradients. For the linear and embedding layers, the runtime overhead increases with the batch size, reaching a factor of up to 5.5× and 49× respectively.

In contrast, enabling DP for RNN-based layers consistently incurs a large (up to 18×) runtime overhead, which decreases as the batch size increases. Opacus's custom RNN-based modules are responsible for most of this overhead, their runtime being up to 11× the runtime of the corresponding `torch.nn` module. As with directly supported `torch.nn` modules, wrapping the custom modules in `GradSampleModule` results in a ~2× slowdown. Fig. 5 compares the runtime of the `torch.nn` module, the corresponding custom module without DP, and the latter wrapped in `GradSampleModule` with DP enabled for the multi-head attention and RNN-based layers.

In practice, Opacus's custom LSTM with DP enabled performs competitively with other DP-SGD frameworks when training with large batch sizes. Recall that on the IMDb dataset, Opacus's

[†]Since $L/C \sim b \leftrightarrow L \sim bC$.

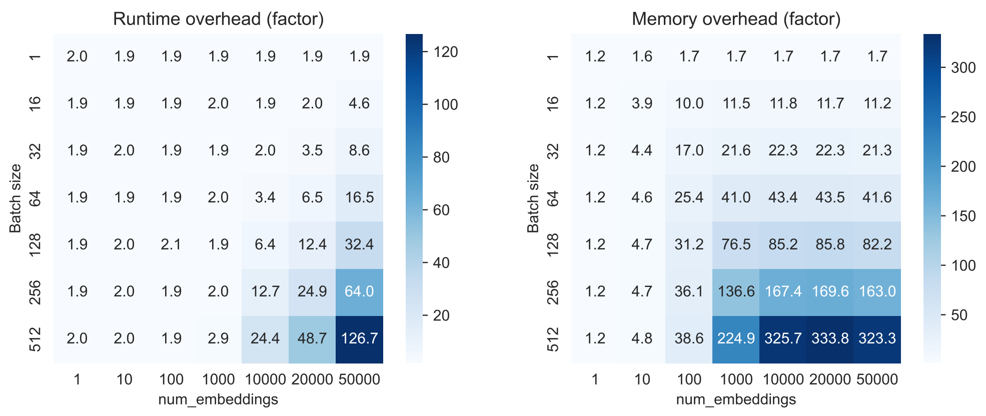


Figure 3: Runtime and peak allocated memory overhead of enabling DP for the embedding layer. In addition to the batch size, we also vary `num_embeddings` and thus, the size of the module L. Left: Runtime overhead (factor). Right: Peak allocated memory overhead (factor). For each value of `num_embeddings` from left to right, $L/C = 0.63, 5, 50, 496, 4{,}951, 9{,}901, 25{,}955$ respectively. Overheads are calculated as in Fig. 2

median per-epoch runtime is only 2.4× the median per-epoch runtime of the custom JAX DP-SGD implementation while avoiding the latter's 101× JIT compilation overhead in the first epoch (see Table 1, Fig. 4).

Peak allocated CUDA memory. For the normalization, multi-head attention, and RNN-based layers, enabling DP with Opacus's `GradSampleModule` results in an up to 1.5× increase in peak allocated memory. In our experiments, L/C is much smaller than the batch size for these layers, hence the relatively constant memory overhead.

For the linear and embedding layers, L/C is substantial compared to the batch size. Hence, as predicted by Eq. (3) and depicted in Fig. 2, the peak allocated memory overhead increases with the batch size, reaching factors of up to 129× and 334× respectively.

Fig. 3 shows how L/C and the batch size affect the runtime and peak allocated memory overhead of enabling DP by example of the embedding layer. Comparing the predicted memory overhead of enabling DP to the measured peak allocated memory overhead for various `num_embeddings` and batch sizes shows that on average, Eq. (3) overestimates the memory overhead of enabling DP by $41\% \pm 17.6\%$ when $L/C \ll b$ and underestimates it by $23.3\% \pm 6.3\%$ when $L/C \gg b$.

For the convolutional layer, the peak allocated memory overhead increases slightly with the batch size. As opposed to the linear and embedding layers, $L/C \ll b$ and the phenomenon weakens as the batch size increases. We attribute this to the comparatively many intermediate computations and additive overheads of the convolutional layer that are not captured in Eq. (2): For the other supported `torch.nn` layers, $M_{\text{non-DP}}$ and M_{DP}, on average, explain $56.5\% \pm 8.7\%$ and $57.5\% \pm 14.9\%$ of the measured peak allocated CUDA memory, whereas for the convolutional layer, $M_{\text{non-DP}}$ and M_{DP} explain only $17.7\% \pm 7.1\%$ and $6.15\% \pm 0.03\%$ of the measured peak allocated memory.

4 Related Work

Gradient Clipping. At the heart of implementing DP-SGD is the need to compute clipped gradients, for which there are several different approaches. A first option, as implemented in Opacus, is to directly compute and clip the per-sample gradients. A second option is to compute only the *norm* of each sample's gradient (in an exact or approximate form), and form the weighted average loss over the batch by weighting samples according to their norm. Typically, each sample's weight is $C/\max(N_i, C)$, where C is the clipping threshold and N_i is the norm of the sample's gradient. The gradient of this loss with respect to the parameters yields the average of clipped gradients. This option was proposed by Goodfellow [6] with exact norms, and was considered more recently along with Johnson-Lindenstrauss projections [3] to compute approximate gradient norms.

Goodfellow's method is based on computing per-sample ℓ_2-norms of the gradients and is restricted to fully-connected layers; more recently, Rochette et al. [17] extended it to CNNs. Lee and Kifer [10] propose computing the *norm* of the per-sample gradients directly, hence doing two passes of back-propagation: one pass for obtaining the norm, and one pass for using the norm as a weight. In Opacus per-sample gradients are obtained in a single back-propagation pass, without performance or accuracy penalties of alternative techniques.

Shortly after an initial version of this work was published, Li et al. [11] generalized the Goodfellow method to handle sequential inputs, making fine-tuning large Transformers under DP computationally efficient. The proposed method, *ghost clipping*, is memory-efficient and has good throughput. In their experiments on large language models, Opacus is as good in memory efficiency and better in throughput than JAX, thanks to some improvements in the implementation. We plan to incorporate such improvements to optimize performance of Opacus further.

Frameworks for differentially private learning. TensorFlow Privacy and PyVacy are two existing frameworks providing implementation of DP-SGD for TensorFlow and PyTorch, respectively. BackPACK [4], another framework for DP-SGD, exploits Jacobians for efficiency. BackPACK currently supports only fully connected or convolutional layers, and several activation layers (recurrent and residual layers are not yet supported). Objax [15] is a machine learning framework for JAX and also provides a DP-SGD implementation. The per-sample gradient clipping in Objax is based on a `vmap` method. It closely resembles the approach implemented in Subramani et al. [19] and we believe benchmark findings are applicable to both implementations. In Section 3 we compare the performance of these frameworks with Opacus.

Finally, we mention alternative approaches to ML training under different notions of privacy and security: using secure hardware to support oblivious training over encrypted data [16], or relying on secure multi-party computation techniques (SMPC) to train over data jointly held by several protocol participants [7].

5 Conclusions

Opacus is a PyTorch library for training deep learning models with differential privacy guarantees. The system design aims to provide simplicity, flexibility, and speed, for maximal compatibility with existing ML pipelines. We have outlined how these design principles have influenced the features of Opacus, and demonstrated that Opacus not only outperforms existing frameworks for training with differential privacy, but performs competitively with custom JIT compiled DP-SGD implementations on a variety of models and datasets.

Opacus is actively maintained as an open source project, supported primarily by the privacy-preserving machine learning team at Meta AI. A number of extensions and upgrades are planned for Opacus in the future, including enhanced flexibility for custom components, further efficiency improvements, and improved integration with the PyTorch ecosystem through projects like PyTorch Lightning.

Acknowledgements

We are extremely grateful to Peter Romov for his substantial and valuable contributions to the Opacus codebase. Opacus also owes a debt of thanks to all of its open-source contributors who continuously helped in the development and maintaining of the Opacus library.

References

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. "Deep Learning with Differential Privacy". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 308–318.

[2] Raef Bassily, Adam Smith, and Abhradeep Thakurta. "Private empirical risk minimization: Efficient algorithms and tight error bounds". In: *55th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2014, pp. 464–473.

[3] Zhiqi Bu, Sivakanth Gopi, Janardhan Kulkarni, Yin Tat Lee, Hanwen Shen, and Uthaipon Tantipongpipat. “Fast and Memory Efficient Differentially Private-SGD via JL Projections”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 34. 2021, pp. 19680–19691.

[4] Felix Dangel, Frederik Kunstner, and Philipp Hennig. “BackPACK: Packing more into Backprop”. In: *International Conference on Learning Representations (ICLR)*. 2020.

[5] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. “Calibrating noise to sensitivity in private data analysis”. In: *Theory of Cryptography Conference*. 2006, pp. 265–284.

[6] Ian Goodfellow. “Efficient per-example gradient computations”. In: *arXiv preprint arXiv:1510.01799* (2015).

[7] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. “CrypTen: Secure Multi-Party Computation Meets Machine Learning”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 34. 2021, pp. 4961–4973.

[8] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. University of Toronto, 2009. URL: `http://www.cs.toronto.edu/~kriz/cifar.html`.

[9] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. 1998. URL: `http://yann.lecun.com/exdb/mnist/`.

[10] Jaewoo Lee and Daniel Kifer. “Scaling up Differentially Private Deep Learning with Fast Per-Example Gradient Clipping”. In: *Proceedings on Privacy Enhancing Technologies* 2021.1 (2021), pp. 128–144.

[11] Xuechen Li, Florian Tramèr, Percy Liang, and Tatsunori Hashimoto. “Large language models can be strong differentially private learners”. In: *International Conference on Learning Representations (ICLR)* (2022).

[12] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. “Learning Word Vectors for Sentiment Analysis”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*. 2011, pp. 142–150.

[13] Ilya Mironov. “Rényi Differential Privacy”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 2017, pp. 263–275.

[14] Ilya Mironov, Kunal Talwar, and Li Zhang. “Rényi Differential Privacy of the Sampled Gaussian Mechanism”. In: *arXiv preprint arXiv:1908.10530* (2019).

[15] Objax Developers. *Objax*. Version 1.2.0. 2020. URL: `https://github.com/google/objax`.

[16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. “Oblivious Multi-Party Machine Learning on Trusted Processors”. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. 2016, pp. 619–636.

[17] Gaspar Rochette, Andre Manoel, and Eric W Tramel. “Efficient Per-Example Gradient Computations in Convolutional Neural Networks”. In: *arXiv preprint arXiv:1912.06015* (2019).

[18] Shuang Song, Kamalika Chaudhuri, and Anand D Sarwate. “Stochastic gradient descent with differentially private updates”. In: *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2013, pp. 245–248.

[19] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. “Enabling Fast Differentially Private SGD via Just-in-Time Compilation and Vectorization”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 34. 2021, pp. 26409–26421.

Appendix

A Micro-Batching

The following code snippet is a naïve way to yield the per-sample gradients through micro-batching.

```
for batch in Dataloader(train_dataset, batch_size):
    all_per_sample_gradients = []
    for x,y in batch:
        y_hat = model(x)
        loss = criterion(y_hat, y)
        loss.backward()

        per_sample_grads = [p.grad.detach().clone() for p in model.parameters()]

        all_per_sample_gradients.append(per_sample_grads)
        model.zero_grad()  # reset p.grad
```

B Vectorized Computation

In accordance with its speed objective, Opacus supports computing per-sample gradients efficiently, in a vectorized manner. This is achieved by deriving a per-sample gradient formula for every layer and transforming it into a form that can be implemented using a single application of the `einsum` operator. Due to space constraints, we discuss this approach only for the `nn.Linear` layer. The implementation details for other layers and other related tutorials can be found in `opacus.ai/tutorials`.

Consider one linear layer with weight matrix W. We omit the bias from the forward pass equation and denote the forward pass by $Y = WX$, where X is the input and Y is the output of the linear layer. X is a matrix of size $d \times B$, with B columns (B is the batch size), where each column is an input vector of dimension d. Similarly, the output matrix Y would be of size $r \times B$ where each column is the output vector corresponding to an element in the batch and r is the output dimension.

The forward pass can be written as follows:

$$Y_i^{(b)} = \sum_{j=1}^{d} W_{i,j} X_j^{(b)},$$

where $Y_i^{(b)}$ denotes the i'th coordinate of the b'th sample in the batch.

In an ML problem, we typically need the derivative of the loss with respect to weights. Correspondingly, in Opacus we need the "per-sample" version of that, which is the per-sample derivative of the loss with respect to the weights W:

$$\frac{\partial L}{\partial z} = \sum_{b=1}^{B} \sum_{i'=1}^{r} \frac{\partial L}{\partial Y_{i'}^{(b)}} \frac{\partial Y_{i'}^{(b)}}{\partial z}.$$

Applying the chain rule above, we can now replace variable z with $W_{i,j}$ and get

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{b=1}^{B} \sum_{i'=1}^{r} \frac{\partial L}{\partial Y_{i'}^{(b)}} \frac{\partial Y_{i'}^{(b)}}{\partial W_{i,j}}.$$

We know from $Y = WX$ that $\frac{\partial Y_{i'}^{(b)}}{\partial W_{i,j}}$ is $X_j^{(b)}$ when $i = i'$, and is 0 otherwise. Continuing the above we have

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{b=1}^{B} \frac{\partial L}{\partial Y_{i'}^{(b)}} X_j^{(b)}.$$

This equation corresponds to a matrix multiplication in PyTorch. In a regular backpropagation, the gradients of the loss function with respect to the weights (i.e., the gradients) are computed for the output of each layer and averaged over the batch. Since Opacus requires computing per-sample gradients, what we need is the following:

$$\frac{\partial L_{batch}}{\partial W_{i,j}} = \frac{\partial L}{\partial Y_{i'}^{(b)}} X_j^{(b)}. \tag{4}$$

More generally, in a neural network with more layers, equation (4) can be written as

$$\frac{\partial L_{batch}}{\partial W_{i,j}^{(l)}} = \frac{\partial L}{\partial Z_i^{(l)(b)}} Z_j^{(l-1)(b)} \tag{5}$$

for every layer l, where $Z_i^{(l)(b)}$ is the activation of the hidden layer l for the b'th element of the batch of the neuron i. We refer to $\frac{\partial L}{\partial Z_i^{(l)(b)}}$ as the *highway gradient*.

We now explain how we compute the per-sample gradient equation (5) in Opacus efficiently. In order to remove the sum reduction to get to the equations (4) and (5), we need to replace the matrix multiplication with a batched outer product. In PyTorch, `einsum` allows us to do that in vectorized form. The function `einsum` computes multi-dimensional linear algebraic array operations by representing them in a short-hand format based on the Einstein summation convention.

For instance, for computing the per-sample gradients for a linear layer, the `einsum` function can be written as `torch.einsum("n...i,n...j->nij", B, A)`, where variables `A` and `B` refer to activations and backpropagations, respectively. In Opacus activations and backpropagations essentially contain what we need for equation (5): using module and tensor hooks in PyTorch, Opacus stores the activations $Z_j^{(l-1)(b)}$ in forward hooks and access the highway gradients $\frac{\partial L}{\partial Z_i^{(l)(b)}}$ through backward hooks. That is how the method `torch.einsum("n...i,n...j->nij", B, A)` implements equation (5) for computing per-sample gradients for a `nn.Linear` layer. To understand the `einsum` expression itself, it is useful to think of it as a generalized version of `torch.bmm` (batch matrix multiplication) for multi-dimensional inputs. For 2D matrices `A` and `B`, `einsum` is equivalent to `torch.bmm(B.unsqueeze(2), A.unsqueeze(1))`. For higher dimensional inputs the idea is the same, while we also sum over extra dimensions.

C Detection of DP Violations

In this section we explain how Opacus can detect whether an operation violates some DP guarantees by applying the following criteria. First, Opacus checks if all layers in the model are supported. Second, Opacus checks for violations that make a model incompatible with differentially private training, which is usually due to one of the two issues: 1) the model tracks some extra information not covered by DP guarantees, or 2) a module is known to do batch-level computations, thus rendering the computation of per-sample gradients impossible.

Some examples: 1) Opacus does not allow models to have batch normalization layers, as they share information across samples; 2) Opacus does not allow `track_running_stats` in instance-normalization layers, as they track statistics that are not covered by DP guarantees.

The above checks in Opacus for DP compatibility are not exhaustive. In particular, Opacus has no way of checking whether the model maintains the independence of the individual samples or tracks extraneous statistics. We plan to investigate ways to address this in the future releases of Opacus.

D Tracking Gradients

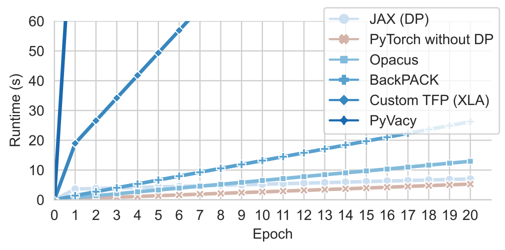
In this section we explain how Opacus makes it easy to keep track of the gradient at different stages of DP training. In the following code snippet, we show how in Opacus we can access intermediate stages of gradient computation throughout training:

```
# model, optimizer and data_loader are initialized with make_private()

for data, labels in data_loader:
    output = model(data)
```

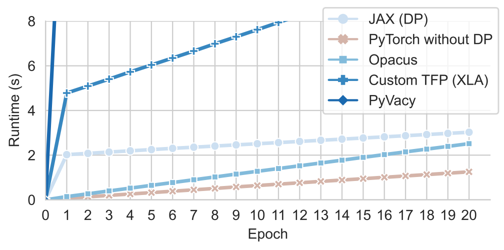
```
loss = criterion(output, labels)
loss.backward()

print(p.grad) # normal gradients computed by PyTorch autograd
print(p.grad_sample) # per-sample gradients computed by Opacus
                     # (no clipping, no noise)

optimizer.step()

print(p.grad_sample) # same as before optimizer.step() - this field is unchanged
print(p.summed_grad) # clipped and aggregated over a batch, but no noise
print(p.grad) # final gradients (clipped, noise added, aggregated over a batch)

optimizer.zero_grad() # all gradients are None now
```

E Additional Experimental Results

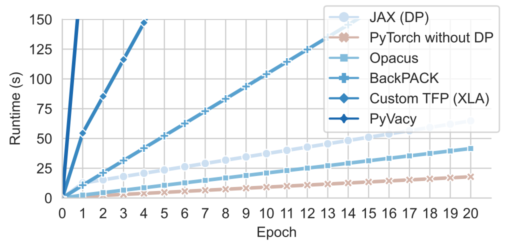
E.1 End-to-end benchmarks



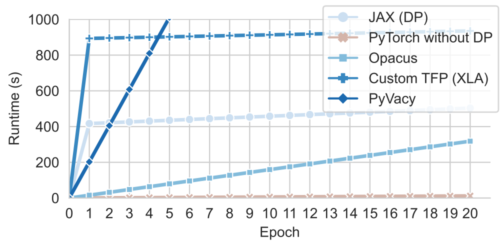
(a) MNIST with CNN



(c) IMDb with Embedding



(b) CIFAR-10 with CNN



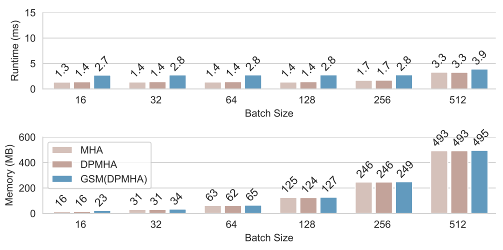
(d) IMDb with LSTM

Figure 4: Cumulative runtime over 20 epochs with batch size 512 for each framework. Using JIT compilation results in a slower first epoch.

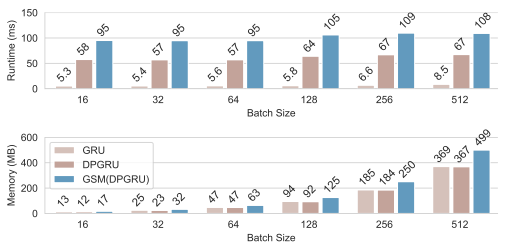
Fig. 4 shows each framework's cumulative runtime over 20 epochs with batch size 512 on each end-to-end model training task. Both JAX (DP) and Custom TFP (XLA) incur a large runtime overhead during the first epoch of up to 101× and 625× the runtime of subsequent epochs respectively due to JIT compilation. If training for relatively few epochs, disabling JIT compilation or using a framework that is optimized to run without JIT may reduce total runtime.

E.2 Microbenchmarks

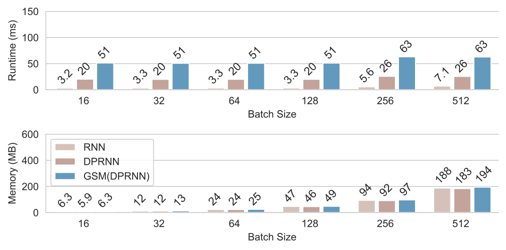
Opacus provides custom implementations for the multi-head attention, RNN, GRU, and LSTM layers, which can be wrapped in `GradSampleModule` to enable training with DP. Fig. 5 compares the runtime and peak memory usage of the `torch.nn` module, the corresponding Opacus module without DP, and the corresponding Opacus module wrapped in `GradSampleModule` with DP enabled for these layers.



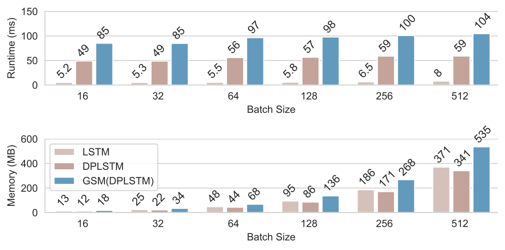
(a) Multi-head Attention



(c) GRU



(b) RNN



(d) LSTM

Figure 5: Comparing the `torch.nn` module, the corresponding Opacus module without DP, and the Opacus module wrapped in `GradSampleModule` with DP enabled for the multi-head attention, RNN, GRU, and LSTM layers. Top: Mean runtime (ms). Bottom: Peak allocated memory (MB).

For RNN-based layers, Opacus's custom modules are responsible for most of the runtime overhead of enabling DP, as they are up to $11\times$ slower than the corresponding `torch.nn` module. Wrapping the custom modules in `GradSampleModule` results in a ~$2\times$ slowdown.

The small peak allocated memory overhead (where applicable) is purely due to wrapping the custom modules in `GradSampleModule` and collecting per-sample gradients, as Opacus's custom modules tend to use slightly less memory than the corresponding `torch.nn` modules.

Table 2 (runtime) and Table 3 (memory) include the raw data used to generate Fig. 2 and Fig. 5. Table 4 includes a breakdown of CUDA memory usage, as well as L/C and $(L/C)/b$ for each layer and batch size.

E.3 Experiment Setup

The exact hardware configurations are listed below. Software versions are listed in Table 5.

End-to-end benchmarks. The end-to-end benchmarks were executed within a virtual environment on a public cloud with an Intel(R) Xeon(R) CPU @ 2.20GHz, NVIDIA A100 SXM4 (40GB VRAM), and 83GB RAM. We created and ran a separate Docker container for each framework. The Docker image source is nvidia/cuda:11.4.2-cudnn8-devel-ubuntu20.04.

Microbenchmarks. The microbenchmarks were executed on a cloud instance running Ubuntu 18.04.5 LTS with an Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz, NVIDIA A100 SXM4 (40GB VRAM), and 1.1TB of RAM. CUDA memory was allocated in block sizes of 512.

For details on the settings for each layer, refer to `https://github.com/pytorch/opacus/blob/main/benchmarks/config.json`.

Layer	Batch size	Runtime (ms) `nn.module`	`DPModule`	`GSM(module)`	Factor (DP)
Conv	16	2.66	n/a	6.18	2.32
	32	4.38		10.9	2.50
	64	8.71		21.7	2.49
	128	14.3		40.0	2.79
	256	27.6		79.0	2.87
	512	54.8		157	2.87
LayerNorm	16	0.27	n/a	0.52	1.92
	32	0.27		0.52	1.91
	64	0.27		0.52	1.91
	128	0.27		0.52	1.91
	256	0.27		0.52	1.90
	512	0.27		0.52	1.91
InstanceNorm	16	0.41	n/a	0.74	1.81
	32	0.41		0.74	1.81
	64	0.41		0.75	1.81
	128	0.73		1.13	1.55
	256	1.43		2.20	1.54
	512	2.77		4.28	1.54
GroupNorm	16	0.30	n/a	0.59	1.94
	32	0.30		0.59	1.94
	64	0.30		0.59	1.94
	128	0.30		0.59	1.95
	256	0.30		0.59	1.95
	512	0.31		0.60	1.91
Linear	16	0.37	n/a	0.65	1.76
	32	0.38		0.66	1.75
	64	0.38		0.66	1.76
	128	0.38		0.68	1.79
	256	0.39		1.08	2.80
	512	0.37		2.05	5.49
Embedding	16	0.25	n/a	0.50	2.01
	32	0.25		0.86	3.45
	64	0.25		1.62	6.52
	128	0.25		3.13	12.4
	256	0.25		6.15	24.9
	512	0.25		12.2	48.7
MHA	16	1.34	1.41	2.72	2.02
	32	1.36	1.43	2.76	2.02
	64	1.37	1.43	2.76	2.02
	128	1.36	1.43	2.77	2.04
	256	1.70	1.72	2.78	1.64
	512	3.29	3.27	3.91	1.19
RNN	16	3.20	20.4	51.3	16.0
	32	3.32	20.0	50.7	15.3
	64	3.34	20.0	50.8	15.2
	128	3.33	20.0	51.1	15.4
	256	5.59	25.8	63.1	11.3
	512	7.08	25.6	62.8	8.87
GRU	16	5.25	57.5	95.0	18.1
	32	5.36	56.8	94.6	17.7
	64	5.61	56.9	94.7	16.9
	128	5.84	63.9	106	18.2
	256	6.64	66.8	110	16.5
	512	8.51	67.2	109	12.8
LSTM	16	5.21	49.0	85.1	16.3
	32	5.32	48.8	84.8	15.9
	64	5.48	56.0	96.5	17.6
	128	5.84	56.6	97.8	16.7
	256	6.55	59.0	101	15.4
	512	8.04	59.1	105	13.0

Table 2: Mean runtime (in milliseconds) for one forward and one backward pass for the `torch.nn` module, the corresponding Opacus module without DP (where applicable), and the corresponding Opacus or `torch.nn` module wrapped in `GradSampleModule` with DP enabled. The factor is the runtime of the layer with DP enabled divided by the runtime of the `torch.nn` module.

Layer	**Batch size**	**Peak allocated CUDA memory (MB)** `nn.module`	`DPModule`	`GSM(module)`	**Factor (DP)**
Conv	16	738	n/a	1157	1.57
	32	929		2312	2.49
	64	1307		4621	3.54
	128	2065		9240	4.48
	256	3582		18479	5.16
	512	6615		36955	5.59
LayerNorm	16	0.03	n/a	0.04	1.32
	32	0.05		0.07	1.33
	64	0.10		0.14	1.33
	128	0.20		0.27	1.33
	256	0.40		0.53	1.33
	512	0.79		1.06	1.33
InstanceNorm	16	6.35	n/a	7.39	1.17
	32	12.7		14.8	1.17
	64	25.4		29.6	1.17
	128	50.7		59.1	1.17
	256	101		118	1.17
	512	203		236	1.17
GroupNorm	16	0.11	n/a	0.14	1.30
	32	0.21		0.27	1.32
	64	0.41		0.54	1.32
	128	0.81		1.07	1.32
	256	1.61		2.14	1.33
	512	3.22		4.27	1.33
Linear	16	3.28	n/a	36.9	11.2
	32	3.42		70.7	20.7
	64	3.68		138	37.6
	128	4.20		273	65.0
	256	5.25		543	103
	512	8.39		1083	129
Embedding	16	24.0	n/a	280	11.7
	32	24.0		536	22.3
	64	24.1		1048	43.5
	128	24.2		2072	85.8
	256	24.3		4122	170
	512	24.6		8217	334
MHA	16	15.7	15.7	23.3	1.48
	32	31.3	31.3	33.9	1.08
	64	62.5	62.3	64.9	1.04
	128	125	125	127	1.02
	256	247	247	249	1.01
	512	493	493	496	1.01
RNN	16	6.27	5.94	6.35	1.01
	32	12.1	11.6	13.1	1.08
	64	23.9	23.5	25.1	1.05
	128	47.3	46.1	48.5	1.03
	256	94.2	92.0	97.3	1.03
	512	188	184	195	1.04
GRU	16	12.7	12.2	16.6	1.31
	32	24.6	23.4	32.0	1.30
	64	47.2	46.8	63.1	1.34
	128	94.2	92.3	125	1.33
	256	185	184	250	1.35
	512	369	368	499	1.35
LSTM	16	13.2	11.5	18.0	1.37
	32	24.7	22.0	34.5	1.40
	64	48.2	43.7	68.3	1.42
	128	94.8	85.9	136	1.44
	256	186	171	268	1.44
	512	371	342	536	1.44

Table 3: Peak allocated CUDA memory (in MB) for one forward and one backward pass for the `torch.nn` module, the corresponding Opacus module without DP (where applicable), and the corresponding Opacus or `torch.nn` module wrapped in `GradSampleModule` with DP enabled. The factor is the peak allocated memory for the layer with DP enabled divided by the peak allocated memory for the `torch.nn` module.

Layer	Batch size	Input (MB)	Labels (MB)	C	Module (L)	L/C	$(L/C)/b$
Conv	16	21.0	16.4	3.36	1.05	0.31	0.02
	32	41.0	33.6	3.38		0.31	0.010
	64	81.9	65.5	3.33		0.32	0.005
	128	164	131	3.33		0.32	0.002
	256	328	262	3.33		0.32	0.001
	512	655	524	3.33		0.32	0.001
LayerNorm	16	0.004	0.004	0.001	0.001	1.33	0.08
	32	0.008	0.008	0.001		1.33	0.04
	64	0.02	0.02	0.001		1.33	0.02
	128	0.03	0.03	0.001		1.33	0.01
	256	0.07	0.07	0.001		1.33	0.005
	512	0.13	0.13	0.001		1.33	0.003
InstanceNorm	16	1.05	1.05	0.20	0.002	0.01	0.001
	32	2.10	2.10	0.20		0.01	0.000
	64	4.19	4.19	0.20		0.01	0.000
	128	8.39	8.39	0.20		0.01	0.000
	256	16.8	16.8	0.20		0.01	0.000
	512	33.6	33.6	0.20		0.01	0.000
GroupNorm	16	0.02	0.02	0.003	0.002	0.67	0.04
	32	0.03	0.03	0.003		0.67	0.02
	64	0.07	0.07	0.003		0.67	0.01
	128	0.13	0.13	0.003		0.67	0.005
	256	0.26	0.26	0.003		0.67	0.003
	512	0.52	0.52	0.003		0.67	0.001
Linear	16	0.03	0.03	0.006	1.05	171	10.7
	32	0.07	0.07	0.006		171	5.34
	64	0.13	0.13	0.006		171	2.67
	128	0.26	0.26	0.006		171	1.34
	256	0.52	0.52	0.006		171	0.67
	512	1.05	1.05	0.006		171	0.33
Embedding	16	0.001	0.007	0.001	8	9259	579
	32	0.001	0.01	0.001		9804	306
	64	0.001	0.03	0.001		9901	155
	128	0.001	0.05	0.001		9901	77.4
	256	0.002	0.10	0.001		9901	38.7
	512	0.004	0.20	0.001		9901	19.3
MHA	16	0.41	0.41	0.08	0.16	2.12	0.13
	32	0.82	0.82	0.08		2.12	0.07
	64	1.64	1.64	0.08		2.12	0.03
	128	3.28	3.28	0.08		2.12	0.02
	256	6.55	6.55	0.08		2.12	0.008
	512	13.1	13.1	0.08		2.12	0.004
RNN	16	0.82	0.82	0.15	0.08	0.53	0.03
	32	1.64	1.64	0.15		0.53	0.02
	64	3.28	3.28	0.15		0.53	0.008
	128	6.55	6.55	0.15		0.53	0.004
	256	13.1	13.1	0.15		0.53	0.002
	512	26.2	26.2	0.15		0.53	0.001
GRU	16	0.82	0.82	0.15	0.24	1.58	0.10
	32	1.64	1.64	0.15		1.58	0.05
	64	3.28	3.28	0.15		1.58	0.02
	128	6.55	6.55	0.15		1.58	0.01
	256	13.1	13.1	0.15		1.58	0.006
	512	26.2	26.2	0.15		1.58	0.003
LSTM	16	0.82	0.82	0.15	0.32	2.11	0.13
	32	1.64	1.64	0.15		2.11	0.07
	64	3.28	3.28	0.15		2.11	0.03
	128	6.55	6.55	0.15		2.11	0.02
	256	13.1	13.1	0.15		2.11	0.008
	512	26.2	26.2	0.15		2.11	0.004

Table 4: Measured sizes (in MB) for the input, labels/output, and the `torch.nn` module for each layer and batch size. These numbers are (almost) identical for the corresponding Opacus module ($\pm 1\%$) and when wrapped in `GradSampleModule`. C is calculated as the size of the input and $2\times$ the size of the labels (to account for model outputs) divided by the batch size.

Software	**Version**
Python	3.8.10 (end-to-end) 3.9.7 (microbenchmarks)
dm-haiku JAX jaxlib	0.0.5 0.2.25 0.1.73
PyTorch	1.10.0
Opacus	1.0.0
BackPACK backpack-for-pytorch	0.1 1.4.0
TensorFlow TensorFlow Privacy	2.7.0 0.7.3
PyVacy	0.0.1 + commit `2e0a9f`

Table 5: Software versions used in the end-to-end and microbenchmarks. The latter only use Python, PyTorch, and Opacus.