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**

Facebook 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, and embedding, right out of the box, and it also provides the means for supporting other user-defined layers. Opacus computes batched per-sample gradients, providing better 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 compare its performance against other frameworks for differential privacy in ML.

1 Background and Introduction

Differential privacy (DP) [4] 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 neighbouring 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) [1] 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. 2, Appendix A for an illustration.) However, mainly 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.

*Equal contribution. Correspondence to `yousefpour@fb.com`.

Preprint. Under review.

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 (sample code to obtain the per-sample gradients using this approach is provided in Appendix B). 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 core component of Opacus is `PrivacyEngine`. It is instantiated with privacy parameters (sample rate and noise multiplier), and keeps track of the "privacy budget" spent so far. `PrivacyEngine` wraps a PyTorch `nn.Module` into a `GradSampleModule`, which adds capabilities for per-sample gradient computations to the backward pass. It also modifies the `optimizer.step()` function to compute noisy gradients and update the privacy budget.

From the user's perspective, all of this is transparent (reflecting the *simplicity* principle in action) and 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)
privacy_engine = PrivacyEngine(
    model,
    sample_rate=batch_size / len(dataset),
    noise_multiplier=noise_multiplier,
    max_grad_norm=max_grad_norm,
)
privacy_engine.attach(optimizer)
# 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 [10, 11]. The `PrivacyEngine` keeps track of how much privacy budget is spent at any given point in time, enabling early stopping and real-time monitoring. Note that Opacus also allows a user to directly instantiate a `PrivacyEngine` targeting an (ϵ, δ) budget: in this case, the engine computes a noise level σ that yields an overall privacy budget of (ϵ, δ).

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 (such as the use of `track_running_stats` in normalization layers).

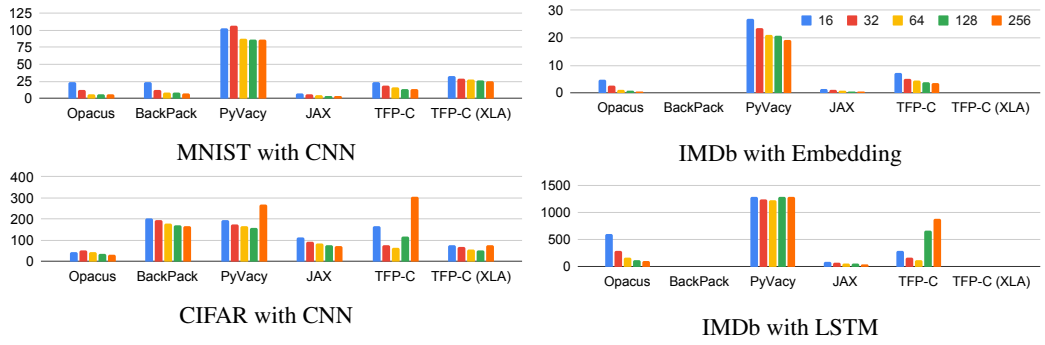

Figure 1: Runtime of the frameworks under different settings (each color representing different batch size). Note that BackPACK does not support LSTM and embedding, hence there are no results for those experiments for BackPACK. An open TensorFlow 2 bug prevents the code from evaluating Custom TFP (XLA) [14, 13].

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 [11].

Vectorized computation. Opacus makes efficient use of hardware accelerators (See Section 3).

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`. In order to maximize the usage of all available memory, Opacus provides an option to decouple physical batch size (which is limited by the amount of memory available) and logical batch size (which is dictated by training utility considerations).

Custom layers. Opacus is flexible as it supports several layer types, including convolutions, LSTMs, multi-head attention, normalization, and embedding layers. When using a custom PyTorch layer (or a standard layer Opacus does not yet support), users can provide a method to calculate per-sample gradients for that layer and register it with a simple decorator provided by Opacus.

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 shuffling.

3 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 implementing its vectorized realization. Due to space constraints, we discuss this approach only for the `nn.Linear` layer (Appendix C). The implementation details for other layers and other related tutorials can be found in `opacus.ai/tutorials`.

We experimentally evaluate runtime performance of Opacus, TensorFlow Privacy, PyVacy, BackPack, and a JAX implementation of DP-SGD. The experiments are based on the benchmarks in [13], but executed with the most current versions of the code of the respective frameworks. The datasets include MNIST, a handwritten digit recognition dataset, CIFAR10, a dataset of small color images, and IMDb [9], a movie review sentiment classification dataset. The models are a CNN with 26,010 parameters for MNIST, a CNN with 605,226 parameters for CIFAR10, and an LSTM network and Embedding network with 1,081,002 and 160,098 parameters, respectively, for IMDb datasets [13]. For TensorFlow Privacy, we choose the custom implementation (referred to as "TFP-C") that uses vectorization and optional XLA-driven JIT compilation, which achieves the best results for this framework [13]. Results are shown in Fig. 1. Since all methods aim to implement essentially the same algorithm for a given privacy level, we do not compare accuracy, but rather focus on running time. All experiments are obtained on Nvidia Tesla K80.

We can see that Opacus and JAX consistently achieve the fastest performance compared to other frameworks for DP-SGD. This is most pronounced when the batch size is larger. PyVacy has the highest running time among all of the frameworks. Both BackPACK and TensorFlow Privacy have comparable performance on the MNIST benchmark, while Opacus has better performance in other experiments. Since BackPACK does not support LSTM and embedding, the corresponding data are omitted.

JAX, a general-purpose framework for high-performance numerical computing, performs extremely well in these experiments. This implementation is based on a hand-crafted custom code for DP-SGD. Opacus, on the other hand, is a library for training deep learning models with differential privacy via a simple and flexible API. Thus, the approach based on hand-crafted JAX code is not compatible with our aim to provide easy and flexible DP versions of existing deep learning pipelines. Moreover, for large-scale ML tasks, it is the compute- and memory-intensive tasks of matrix multiplication and other tensor operations that ultimately form the real bottleneck for any framework. Finally, we believe that the picture will look different when similar functionalities in JAX (e.g., `vmap`) are implemented in Pytorch and Opacus is be able to leverage them.

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 weighing 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 [5] with exact norms, and was considered more recently along with Johnson-Lindenstrauss projections [2] to compute approximate gradient norms.

The method in [5] is based on computing per-sample ℓ_2-norms of the gradients and is restricted to fully-connected layers; more recently, Rochette *et al.* [12] extended it to CNNs. Lee and Kifer [8] 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.

Existing frameworks. There are a few other frameworks for differentially private learning. TensorFlow Privacy and PyVacy are two existing frameworks providing implementation of DP-SGD for TensorFlow and PyTorch, respectively. BackPACK [3], 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). In Section 3 we compare the performance of these frameworks with Opacus.

Last, we point to some efforts to perform ML training under different notions of privacy and security: the Myelin system makes use of secure hardware to add noise centrally [6], while Crypten [7] does training in a secure multi-party setting, without explicit noise addition.

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 it achieves best-in-class performance on a battery of ML tasks compared to other DP training frameworks.

Opacus is actively maintained as an open source project, supported primarily by the privacy-preserving machine learning team at Facebook. 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 PyTorch ecosystem through projects like PyTorch Lightning.

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* (Oct. 2016).

[2] Zhiqi Bu, Sivakanth Gopi, Janardhan Kulkarni, Yin Tat Lee, Judy Hanwen Shen, and Uthaipon Tantipongpipat. "Fast and Memory Efficient Ddifferentially Private-SGD via JL Projections". In: *arXiv preprint arXiv:2102.03013* (2021).

[3] Felix Dangel, Frederik Kunstner, and Philipp Hennig. "BackPACK: Packing more into backprop". In: *arXiv preprint arXiv:1912.10985* (2019).

[4] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. "Calibrating noise to sensitivity in private data analysis". In: *Theory of cryptography conference*. Springer. 2006, pp. 265–284.

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

[6] Nick Hynes, Raymond Cheng, and Dawn Song. "Efficient Deep Learning on Multi-Source Private Data". In: *CoRR* abs/1807.06689 (2018). arXiv: `1807.06689`.

[7] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. *CrypTen: Secure Multi-Party Computation Meets Machine Learning*. 2021. arXiv: `2109.00984 [cs.LG]`.

[8] Jaewoo Lee and Daniel Kifer. "Scaling up Differentially Private Deep Learning with Fast Per-Example Gradient Clipping." In: *Proc. Priv. Enhancing Technol.* 2021.1 (2021), pp. 128–144.

[9] 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*. 2011, pp. 142–150.

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

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

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

[13] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. "Enabling Fast Differentially Private SGD via Just-in-Time Compilation and Vectorization". In: *arXiv preprint arXiv:2010.09063* (2020).

[14] Nicholas Vadivelu. *XLA compilation does not work with Embeddings Layer*. `https://github.com/tensorflow/tensorflow/issues/43687`. 2020.

Appendix

A Representation of the DP-SGD algorithm and its implementation in Opacus

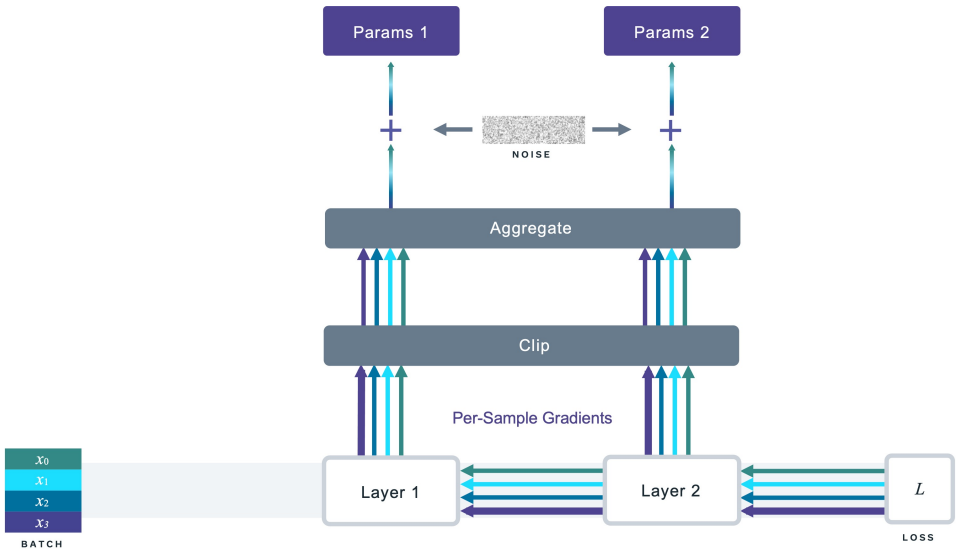

Figure 2: Pictorial representation of the DP-SGD algorithm. The single-colored lines represent per-sample gradients, the width of the lines represent their respective norms, and the multi-colored lines represent the aggregated gradients.

B 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
```

C Vectorized Computation for `nn.Linear`

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 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:

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

More generally, in a neural network with more layers, equation (1) can be written as

$$\frac{\partial L_{batch}}{\partial W_{i,j}^{(l)}} = \frac{\partial L}{\partial Z_i^{(l)(b)}} Z_j^{(l-1)(b)} \tag{2}$$

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 (2) in Opacus efficiently. In order to remove the sum reduction to get to the equations (1) and (2), 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` is for evaluating the Einstein summation convention on the operands; it allows computing many 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 store what we need for equation (2): 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 (2) for computing per-sample gradients for a `nn.Linear` layer.