On the Time and Space Overhead of Differentially Private Stochastic Gradient Descent Training

Shaoxuan Zheng, Yuchong Zhang, Jimmy Lin

Abstract

Despite its simplicity, the Differentially Private Stochastic Gradient Descent (DP-SGD) algorithm has significant running time and memory overhead compared to the non-private SGD algorithm. We investigate the possible causes for this problem and lay out some potential solutions.

1 Introduction

Stochastic gradient descent (SGD) has proven to be an effective algorithm for training large neural networks. However, SGD requires a large amount of training data, which naturally raises the concern for data privacy. In both industry and academic research fields, the training data-set may contain user specific data, which can contain sensitive information and should not be released. An adversary might even be able to infer such sensitive information from the trained model without direct accessing the training set. For example, Fredrikson et al. illustrated how to attack a facial recognition system to recover facial images by inversing the model [1].

Differential Privacy (DP) is a popular definition for privacy. The intuition is that an algorithm is differentially private if the adversary is not able to tell whether or not the information for a specific user is used in the computation. Authors of [2] proposed the differentially private SGD (DP-SGD) algorithm, the privacy guarantee of which was proved via an accounting method. However, existing implementations of DP-SGD in frameworks such as PyTorch, Tensorflow, and JAX have significant running time and memory overhead compared to its non-private counterpart, which motivates us to further investigate this problem more deeply.

In this paper, we:

- Conduct detailed benchmark experiments of implementations of DP-SGD in multiple frameworks, and compare its memory and running time overhead to normal SGD training.
- Carefully analyze the possible causes for these overheads, and propose some possible solutions and directions to solving the problem.

2 Background

2.1 Differential Privacy(DP)

Let $\epsilon, \delta > 0$, and let D, R be any two sets. We say a random mechanism $\mathcal{M} : D \to R$ is (ϵ, δ)-differentially private if for any two adjacent inputs $d, d' \in D$ and subset $S \in R$, we have

$$P(\mathcal{M}(d) \in S) \leq e^{\epsilon} P(\mathcal{M}(d') \in S) + \delta$$

where the meaning of "adjacent" depends on the context of the application at hand. In our case, d, d' would be two training sets and we say they are adjacent if they differ by a single training example.

Intuitively, we want to make sure that if we train the same model from scratch on two adjacent training sets d, d', the results should be roughly "indistinguishable" so that one cannot infer from the resulting models any information about the data point that makes d and d' distinct.

2.2 Stochastic Gradient Descent(SGD)

SGD is an algorithm for training a neural network, where at each iteration, one randomly samples a mini-batch from the training set and computes the gradient of the loss function on that batch, then performs gradient descent. The idea is that if we assume the data points are i.i.d., then the batch gradient would be an unbiased estimate of the gradient on the whole training set. SGD itself does not provide any privacy guarantee, so researchers have combined it with differential privacy, which we describe below.

2.3 Differentially Private Stochastic Gradient Descent(DP-SGD)

Compared to normal SGD, the DP-SGD algorithm proposed in [2] consists of computing the gradient on each training example within a batch, clipping the gradients by some gradient bound, and adding Gaussian noise.

DP-SGD:[2]:
Input: Training data $\{ X_1, , , X_n \}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.
Initialize θ_0 randomly
for t $\in$ {T} **do**
Take a random sample L_t with sampling probability $\frac{L}{N}$
Compute gradient
For each i $\in L_t$, compute $g_t(x_i) \leftarrow \mathcal{L}(\theta_t, x_i)$
Clip gradient
$\overline{\mathbf{g_t}}(x_i) \leftarrow \mathbf{g_t}(x_i)$ / $\max(1, \frac{\|\mathbf{g_t}(x_i)\|_2}{C})$
Add noise
$\widetilde{\mathbf{g_t}} \leftarrow \frac{1}{L}\sum_i(\overline{\mathbf{g_t}}(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I}))$
Descent
$\theta_{t+1} \leftarrow \theta_t - \eta_t \widetilde{\mathbf{g_t}}$
Output θ_T and compute the overall privacy cost (ϵ, δ) using a privacy accounting method

3 Motivation

3.1 Computational Resources

DP-SGD training usually involves a large DNN model and a large amount of training data. GPU accelerators are expensive and access is usually shared among lots of users. Moreover, DP-SGD is often applied in the context of federated learning, where the computations are performed happens on mobile devices with limited computational power. Thus, there is demand for highly optimized implementations of DP-SGD.

3.2 Time Overhead

If per-sample gradient computation can be fully parallelized, then DP-SGD should not have a significant running time slowdown compared with the baseline SGD training because we expect clipping and noising to be fast. In reality, according to various profiling experiments on DP-SGD, there are slowdown ratios between 2X and 100X across different frameworks on the MNIST data-set. Although there are extra workloads in DP-SGD, the slowdown should not be so significant, so we suspect that the implementations of DP-SGD are not fully optimized or parallelized.

3.3 Space Overhead

We also profiled the maximum batch size allowed for both DP-SGD and the baseline SGD training on various models, and find that in most cases, DP-SGD consumes much more memory compared to SGD, with a ratio of up to 1 : 15. Apart from the obvious downside of requiring more computational resources, the memory overhead could also be detrimental to the quality of the trained model since it has been shown that DP-SGD training often requires a large batch size to be effective.

4 Related Work

4.1 Acceleration with Just-in-Time Compilation and Vectorization

Pranav Subramani et al have worked on hardware support for accelerating the training of DP-SGD [3]. They tested the performance of DP-SGD on multiple frameworks, including JAX, which supports vectorization, just-in-time compilation and static graph optimization. They also rebuilt the core part of TensorFlow Privacy with the support of effective vectorization and XLA compilation, which provides memory and running-time improvements.

For the experiments, they ran tests on the Adult data-set with logistic regression, experiments on MNIST and CIFAR10 data-set with CNN, experiments on IMDb data-set with Embedding network and LSTM network. The framework and libraries in which they implemented the experiments include JAX, their customized TFP, Chain-Rule-Based Per-Example Gradients [4, 5], BackPACK [6], Opacus [7], PyVacy [8] and TensorFlow Privacy [9]. They also run all experiments across different batch sizes: 16, 32, 64, 128, 256.

The experiment results can be summarized as follows:

- In terms of absolute running time for private training, JAX is always the fastest, with a slowdown ratio less than 2X compared with the non-Private version. TensorFlow Privacy and PyVacy are always the slowest, with running time being at least 10 times that of the non-private training.
- For their customized version of TensorFlow Privacy, the improvements are salient, with the execution being accelerated tens of times compared with the vanilla TensorFlow Privacy.

Based on the result for JAX and their customized Tensorflow Privacy, it seems that JIT and XLA compilation can speed up private training significantly.

4.2 Ghost Clipping

Bu et al extend the algorithm of ghost clipping to convolution layers [10]. With this, plus prior works that extend ghost clipping to linear layers [4], layer-norm layers and embedding layers [11], makes it possible to apply ghost clipping to DP-SGD training for common neural network architectures.

Since DP-SGD requires clipping per-sample gradients, previous works naturally calculated per-sample gradients before the clipping step. However, it turns out that the calculation of per-sample gradient may not always be necessary or computationally optimal. The benefit of DP-SGD with ghost clipping is that we can get per-sample norms without calculating the potentially expensive per-sample gradient. More specifically, by leveraging certain intermediate quantities stored in the computational graph during the forward pass, one can compute the per-sample gradient norms. After getting per-sample gradient norms, one can re-weight the loss by the clipping thresholds of per-sample gradients, then conduct a second back-propagation to derive the clipped gradients.

Authors of [10] did run-time complexity analysis for per-sample gradient and ghost clipping, and derived a explicit condition on when to apply per-sample gradient and when to apply ghost clipping. They call this algorithm "mixed ghost clipping".

Their result shows that with models like VGG, ResNet and Vision Transformers, DP-SGD training with ghost clipping only raise around 1% to 10% memory overhead and < 2x slowdown compared to non-private training. For a specific example, when training with VGG19 on CIFAR10, their approach is 3x faster than Opacus and has 18x larger maximum batch size.

5 Details Of Profiling Experiments

5.1 Overall Timing

First, we implement VGG and DP-SGD in TensorFlow, PyTorch and Jax. The training uses the MNIST data-set and batch size 256, with RTX 1080Ti GPU support. The results are as follows:

Framework	Public Batch Time	Private Batch Time	Slowdown Ratio
TensorFlow	0.736s	3s	4 times
PyTorch	0.006s	0.007s	1.2 times
JAX	0.00076s	0.00414s	5.4 times

As we can see from the table, the fastest frameworks are PyTorch and JAX, so we decide to decompose the execution of DP-SGD in PyTorch and JAX to find out where the overhead comes from. We examine the EPOCH training time of DP-SGD, non-private SGD, SGD with per-sample gradient but without clipping and noising using 4 different models and MNIST dataset in JAX and PyTorch. The result details are

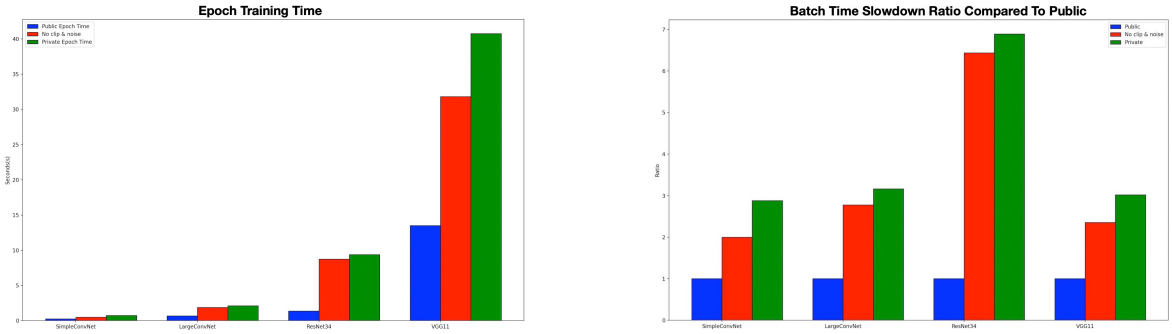


Figure 1: JAX **Left**: EPOCH run-time, **Right**: Run-time Ratio

As we can see from the second image, across the 4 models, the smallest slowdown ratio for DP-SGD in JAX is 3 times, while the max ratio is almost 7 times.
The result here demonstrates that even in the most optimized framework, private training is still much slower than non-private training, and it can be inferred that the overhead is from per-sample gradient computation rather than clipping and noising.

Results from PyTorch point to similar conclusions and are illustrated below:

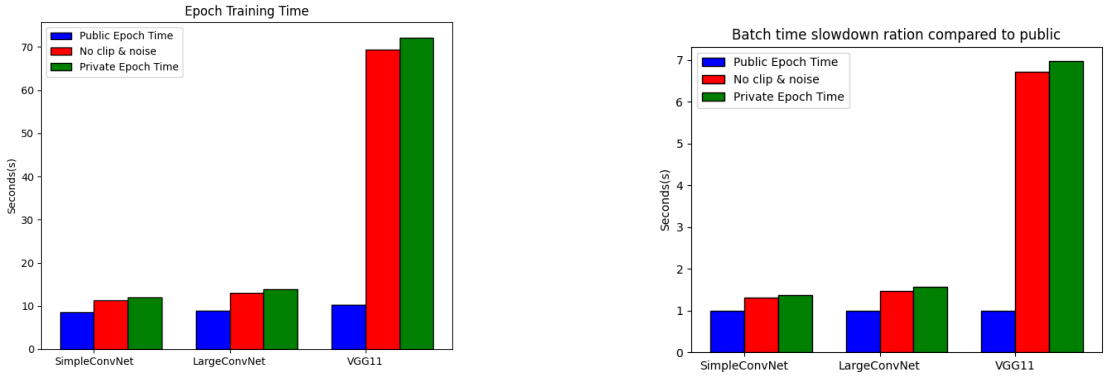


Figure 2: PyTorch **Left**: EPOCH run-time, **Right**: Run-time Ratio

To summarize: for both PyTorch and JAX, private training incurs significant overhead over public training when model size is large, and the main overhead is due to per-sample gradient computation.

5.2 Customized Time Profiling for DP-SGD in JAX

In order to speed up the training, We tried several profiling tools to decompose the training time in order to determine the main cause of the overhead. The profiling tools we tried include cProfile module, flameChart and Tensorboard, but all of them show large discrepancy compared to absolute time counting, which means note the start and end time, and take the difference. So we built our customized way of counting time.

As illustrated in Algorithm 1, DP-SGD consists of 3 steps: (1)Per-sample gradient (2)Clipping per-sample gradients (3)Noising on averaged clipped gradients. So our method is to first count the

total run-time with all 3 components included, then remove each one of them, and then take the difference, which is the execution for the corresponding removed section. The result for JAX is shown below:

Model	Private Time	Batch Gradient	Without Clipping	Without Noising
SimpleConvNet	0.72s	0.4s	0.56s	0.65s
LargeConvNet	2.12s	1.2s	1.93s	1.97s
Simplified ResNet-34	9.50s	5s	8.9s	9.37s
VGG11	40.75s	22.5s	32s	40.27s

After getting above result, we can take the difference between the time after removing each part and private time to get the overhead for (1) Per-sample gradient (2) Clipping (3) Noising

Model	Private Time	Per-sample Gradient	Clipping	Noising
SimpleConvNet	0.72s	0.32s	0.16s	0.07s
LargeConvNet	2.12s	0.92s	0.19s	0.15s
Simplified ResNet-34	9.50s	4.5s	0.6s	0.13s
VGG11	40.75s	18.25s	8.75s	0.48s

We can see that the overhead is mainly from per-sample gradient, while the time for clipping and noising are negligible.

5.3 Customize Time Profiling On PyTorch

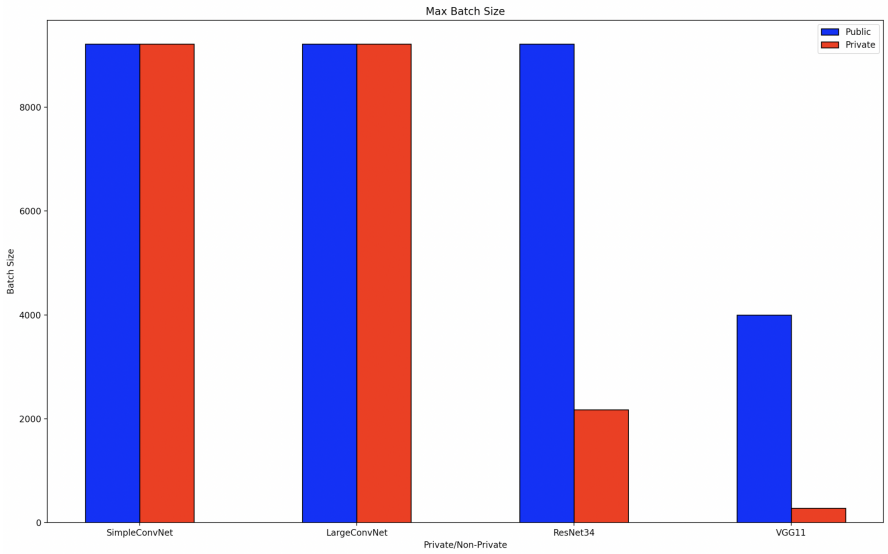
Model	Private Time	No Clipping and no noising	Without Clipping	Without Noising
SimpleConvNet	11.99s	11.37s	11.28s	11.89s
LargeConvNet	13.94s	13.08s	13.36s	13.11s
VGG11	72.16s	69.45s	70.62s	70.01s

The PyTorch timing results also point to the same conclusion. As we can see, removing clipping and/or noising has negligible effect on the per epoch training time. From this, and the fact that private training is significantly slower than public training, we conclude that per-sample gradient is the main overhead for runing time.

5.4 Space Overhead In JAX

In order to compare the memory usage between private and non-private SGD training, we first tried to visualize the NVIDIA GPU change during training, and find that JAX pre-allocates 90% of the available memory, leading to same result between private and non-private training. Then we found flags on how to turn this feature off, details can be found here. After turning this feature off, the peak memory usage is still similar, and we realize that peak memory usage may not be a reasonable way to compare memory usage, so we decide to compare maximum batch size allowed in private and non-private training.

We use the same 4 models as from Time Overhead part, and training data-set is MNIST, with RTX 1080Ti GPU support.



It can be seen that for the 2 larger models, there are substantial differences in maximum batch size between private and public training. This is especially obvious with VGG11, where the max non-private batch size is almost 15 times larger than that for private version, which shows the enormous space overhead for private training.

As for the two small models, the max batch sizes are all the same, which is kind of bizarre. We believe the reason is because of the memory limitation from GPU, if we run the experiment on a more advanced GPU with more memory, there should be a similar gap as shown in the case of larger models.

5.5 Space Overhead In PyTorch

We tried several methods to investigate the memory overhead in PyTorch training. First, we observed a large discrepancy in maximum batch size between public training and private training when a large model (VGG11) is used:

Model	Public max batch size	Private max batch size
VGG11	8625	100

This motivates us to perform further profiling on the memory usage of private training in order to identify the main overhead. We measured the amount of memory that is allocated for both public and private training. When the model is VGG11 and batch size is 100, we have:

Public	Private
1425MB	10261MB

We then broke down these memory usage into 4 components:

- model: memory usage of storing model parameters.
- gradient: memory usage of storing gradients.

- activation: memory usage of storing activations computed during the forward pass.
- data: memory usage of storing batches of data points.

Figure 3: memory breakdown **Left**: public, **Right**: private

These breakdowns imply that the storage of per-sample gradient is the main overhead for the increased memory usage of private training. To further illustrate the difference in memory usage behaviour for private and public training, we also include the tensorboard profiling results here:

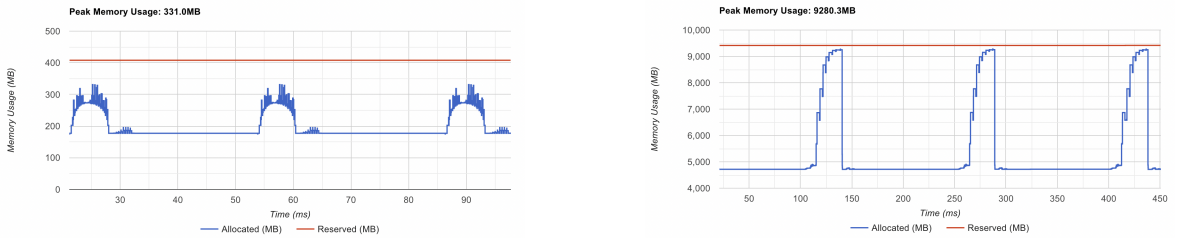


Figure 4: memory usage pattern **Left**: public, **Right**: private

6 Methods We Tried And Further Direction

6.1 Optimize Noising

Based on our earliest profiling results using cProfile, noising incurs a surprisingly large overhead, which takes around half of the training time, so we spent lots of time on optimizing noising.

The first way we tried is to generate more noise by a single function call. The insight is that, we find if the size of Gaussian noise that is passed to Gaussian function is enlarged by tens of times, the time for generating larger noise is roughly the same. So our idea is to generate the noises for next n iterations, let's say n=10, and for each iteration, instead of calling the noise-generation function once, we directly pass in the pre-generated noise, which ideally should save some time.

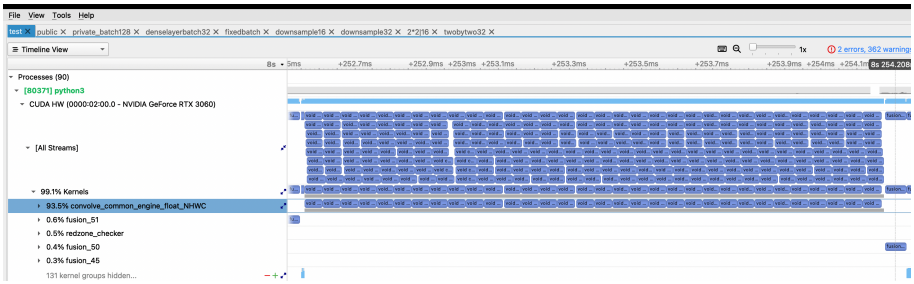
However, the problem comes when passing the pre-generated noise into each iteration. Since the pre-generated noises are stored in a length 10 array, so the naive way of passing in noise is by direct indexing, eg noise[0]. But the absolute timing result shows that the EPOCH time is prolonged by around 4 times compared to private training, and the counting result shows that the indexing part takes much more time than expected, which leads to the slowdown.

Another approach we tried is to split the array into 10 pieces using a splitting function, and for each EPOCH, passing a piece of noise into private training. This approach is better than direct indexing, but still leads to twice EPOCH training time compared to private training. And we don't find any other way of splitting pre-generated noise, so we have to abnegate this approach.

6.2 Profile Kernels Of DP-SGD

If the calculation of per-sample gradient is fully parallelized, then the gradient calculation part should not raise so much overhead, so we are skeptical whether the implementation of per-sample gradient is fully parallelized in the kernel level.

The profiling approach we tried is the NVIDIA Nsight System, which can be found here. The profiling result shows that 93.5 % of the kernel calls are convolutions, with kernel name "convolve_common_engine_float_NHWC". A snippet of profiling result is attached below



The total number of "convolve" kernels is exactly the same as the batch size, which is 256 in this case. All the kernels are divided into 8 streams, with 32 kernels for each stream. But by these results, we are unable to conclude that the implementation of per-sample gradient is not fully parallelized.

Since we have made sure that most of the overhead is from per-sample gradient, hence writing customized kernels for per-sample gradient might be a good direction to head on, and possible documents can be found here

6.3 Mixed Ghost Clipping

Recently, Bu et al. proposed Mixed Ghost Clipping as a potential solution to the increased memory and time overhead incurred by private training [10]. We ran some experiments on this method to determine the degree to which it solves the problem. As in the previous section, the following results is based on experiments ran with VGG11.

First, we measured the per-epoch training time of Mixed Ghost Clipping using batch size = 100.

Public	Private	Mixed Ghost Clipping
9.06s	71.17s	27.28s

As we can see, mixed ghost clipping does significantly reduce the run time for private training, although there is still a large gap between it and public training.

Next, we tested the maximum batch size of Mixed Ghost Clipping.

Public	Private	Mixed Ghost Clipping
8625	100	3500

Similar as before, mixed ghost clipping increases the maximum batch size by a lot, although not to the level of public training.

Finally, we directly measured the peak memory usage of Mixed Ghost Clipping using the maximum batch size 3500:

Public	Mixed Ghost Clipping
6193MB	8055 MB

We suspect that the discrepancy shown above is largely due to the storage of intermediate results in the process of computing per-sample gradient norms by Mixed Ghost Clipping, in which case writing customized kernels with better memory management could be a direction.

7 Conclusion

In this paper, we provided detailed profiling results regarding the time and space overhead of DP-SGD implementations in PyTorch and JAX. We find that most of the overhead is from per-sample gradient no matter what the model is. We also tried several approaches to speed up the training. In the end, we point out a direction that writing customized kernels for per-sample gradient may be a good solution.[1]

References

[1] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Martin 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*. ACM, oct 2016.

[3] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. Enabling fast differentially private SGD via just-in-time compilation and vectorization. *CoRR*, abs/2010.09063, 2020.

[4] Ian Goodfellow. Efficient per-example gradient computations, 2015.

[5] Gaspar Rochette, Andre Manoel, and Eric W. Tramel. Efficient per-example gradient computations in convolutional neural networks. 2019.

[6] Felix Dangel, Frederik Kunstner, and Philipp Hennig. Backpack: Packing more into backprop, 2019.

[7] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in pytorch, 2021.

[8] Tim Gianitsos Chris Waites. Pyvacy: Privacy algorithms for pytorch.

[9] Andrew Galen Nicolas Papernot and Steven Chien. Tensorflow privacy.

[10] Zhiqi Bu, Jialin Mao, and Shiyun Xu. Scalable and efficient training of large convolutional neural networks with differential privacy, 2022.

[11] Xuechen Li, Florian Tramèr, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners, 2021.

[1] Scripts for all of our experiments can be found at https://github.com/yc7z/ml-privacy-exp