Breakdown Of Time And Space Overhead In Differentially Private Stochastic Gradient Descent Training

Shaoxuan Zheng, Yuchong Zhang, Jimmy Lin

September 10, 2022

Abstract

The definition of differential privacy has been combined with stochastic gradient descent training for years, while most people put their attention on how to improve the accuracy after privacy mechanism, this paper focus on the run-time and space overhead that is brought by the privacy mechanism and lay out a couple of possible directions that may allay the pain about run-time and space from introducing differential privacy into stochastic gradient descent training.

1 Introduction

Although Stochastic gradient descent(SGD) training has been proven to be useful in multiple areas such as image classification, computer vision and self-driving etc, the concern about privacy of train data-set has raised these years. In both industry and academic research fields, the train data-set usually contains user specific data, which is sensitive and should not be released. For example, transnational corporation may want to train a model to better service their customer, but the train data-set for the model comes from their customer, and may even contain vulnerable information from each country, hence, raw data must not be used to train the model directly and privacy must be retained. In some cases, an adversary may be able to steal some of the information from train data-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 is an algorithm that provides strong privacy guarantees for aggregated data. Roughly speaking, 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. The combination of Differential Privacy and Stochastic Gradient Descent (DP-SGD) is able to mitigate the concern on privacy for training that involves SGD. However, current framework such as Tensorflow, PyTorch and Jax doesn't fully optimize the implementation of DP-SGD due to a variety of reasons, in this paper, we will talk about :

- By comparing to raw SGD training, we demonstrate and analysis the Time and Space Overhead for DP-SGD in Tensorflow, PyTorch and JAX
- Possible solutions and directions to solve 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)

Differentially Private Stochastic Gradient Descen (DP-SGD) is one of the most well-known strategy to alleviate the concern from privacy. The state of the art algorithm is illustrated as below:
Algorithm1[12]:
Input: Examples $\{ X_1, , , , X_n \}$, 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 $g_t(x_i) \leftarrow$ L(θ_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

The training of DP-SGD usually involves a large DNN model, such as ResNet or VGG, and a colossal amount of training data-set. GPU accelerators are expensive and the access is usually shared by lots of users, with the limitation of computational resources, the more optimized the implementation is, the faster the training is going to finish, hence take less amount of computational resources. Besides, the application of DP-SGD is always associated with federated learning, where the computation usually happens on mobile devices, as the computation power of chip on mobile devices are limited by its size, and there is usually not too much battery left for the training, hence, the demand of highly optimized DP-SGD implementation is inevitable.

3.2 Time Overhead

Ideally, DP-SGD shouldn't cause a severe run-time slowdown compared with its baseline SGD training because if per-sample gradient can be fully paralleled, clipping and noising should be fast. However, we do a run-time profile on DP-SGD, and the result shows that there are slowdown ratios between 2X to more than 100X across different frameworks on MNIST data-set. Although there are extra workload in DP-SGD, the slowdown shouldn't be so heavy, which implies that the implementation for DP-SGD is not fully optimized or not fully paralleled. In real life application, under the limitation of computation resources, the un-optimization will result in slower convergence of the model. So we should pay attention to this problem.

3.3 Space Overhead

We also profile the maximum batch size allowed for both DP-SGD and its baseline SGD training on various models, and find that in most cases, DP-SGD consumes more spaces compared to SGD, with up to a ratio of $\frac{1}{15}$ compared to baseline. Since the computation resources are limited, in most cases, the user wants to take advantage of all the computation resources, but the discrepancy of max batch size will degenerate the slowdown of run-time, hence prolong the training time of the model.

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[2]. They test the performance of DP-SGD on multiple frameworks, including JAX, which supports vectorization, just-in-time compilation and static graph optimization. They also rebuild the core part of TensorFlow Privacy with the support of effective vectorization and XLA compilation, which provides tremendous memory and run-time improvements.
For the experiments, they run 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 are also various, including JAX, their customized TFP, Chain-Rule-Based Per-Example Gradients[3,4], BackPACK[5], Opacus[6], PyVacy[7] and TensorFlow Privacy[8]. They also run all experiments across different batch sizes, which are 16, 32, 64, 128, 256
As for the experiment results:

- For absolute running time, JAX is always the fastest one, with slowdown ratio less than 2X compared with Non-Private version. TensorFlow Privacy and PyVacy are always the slowest, with running time being at least 10 times compared to JAX generally.
- For their customized version TensorFlow Privacy, the improvements are salient, with the execution being accelerated tens of times compared with raw TensorFlow.

Based on the result for JAX, it shows that the framework that supports JIT and XLA compilation is always the most optimized framework across all widely used frameworks. The conspicuous improvements from their customized TensorFlow Privacy also shows that JIT and XLA compilation tools are able to speed up training with DP a lot.

4.2 Ghost Clipping

Bu et al extend the algorithm of ghost clipping from linear algebra to convolution layers[9], plus prior work with extensions to linear layer[10], layernorm layer and embedding layer[11], they are able to embed ghost clipping into DNN training.
Since DP-SGD requires to clip on per-sample gradient, hence the calculation of per-sample gradient is always needed in previous arts. However, it turns out that the calculation of per-sample gradient is not always necessary and not optimal. The benefit of DP-SGD with ghost clipping is that we can get per-sample norm without calculating the expensive per-sample gradient.
After getting per-sample gradient, they can re-weight the loss to satisfy the clipping threshold, then conduct a second back-propagation to derive the weighted gradient, which cost some extra time.
Besides above contribution, they also did run-time complexity analysis for per-sample gradient and ghost clipping, and derive a concrete condition on when to apply per-sample gradient and when to apply ghost clipping, they call this algorithm as "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, they approach is 3x faster than Opacus, which is the state-of-the-art, and with 18x maximum batch size. And all the above achievements have no impact accuracy

5 Details Of Profiling Result

5.1 Overall Timing

First we run experiments on TensorFlow, PyTorch and Jax with MNIST data-set, batch size=256, with RTX 1080Ti GPU support. The results are shown as below:

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 on PyTorch and JAX to find out where is the overhead from. We examine the EPOCH training time of DP-SGD, non-private SGD, SGD with per-sample gradient on 4 different models on MNIST dataset using JAX and PyTorch. The result details are

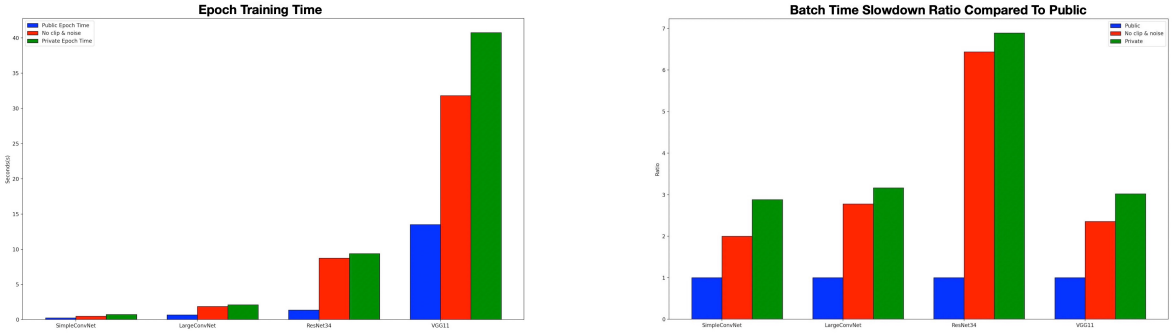


Figure 1: JAX **Left**: EPOCH run-time, **Right**: Run-time Ratio

As we can see from the second image, across the 4 models, the least 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, because the only difference between blue bar and red bar is that, blue bar take batch gradient, while red bar takes per-sample gradient and then take the average.

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

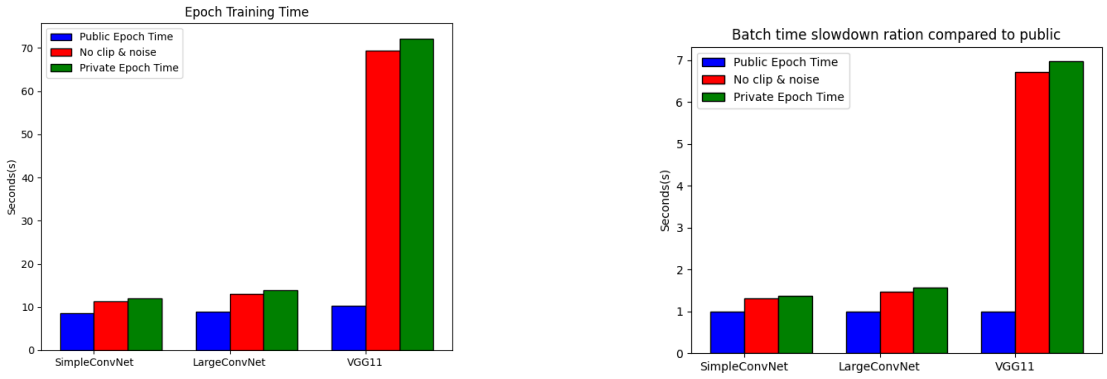


Figure 2: PyTorch **Left**: EPOCH run-time, **Right**: Run-time Ratio

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 Customize Time Profiling On JAX

In order to speed up the training, We tried a couple of profiling tools to decompose the training to see where is the overhead from, 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 build our customized way of counting time. As mentioned in Algorithm 1, DP-SGD is mainly composed of (1)Per-Sample Gradient (2)Clip on Per-Sample gradient (3)Noising on averaged clipped gradient, 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.

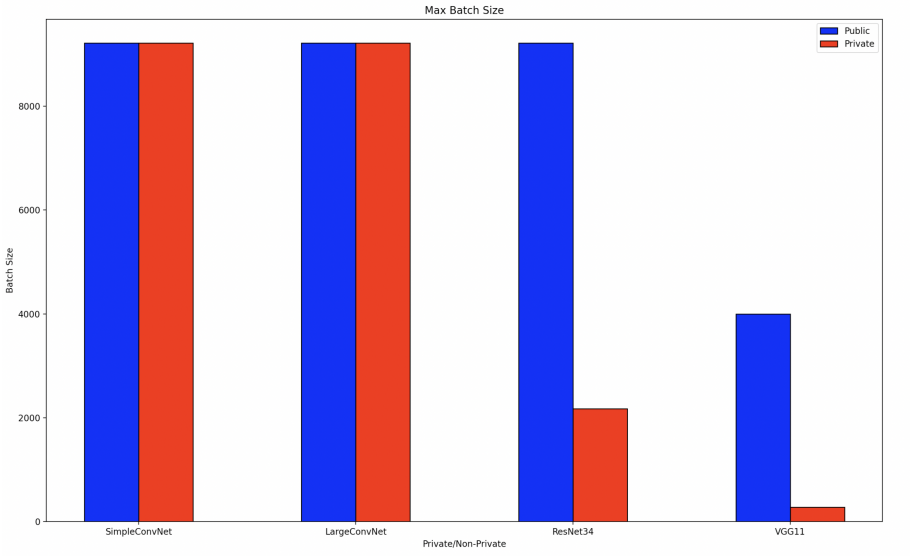
5.3 Customize Time Profiling On PyTorch

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 try to visualize the NVIDIA GPU change during training, and find that in JAX, it will pre-allocate 90% of available memory, hence lead to same result between private and non-private version. Then we find 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 train data-set is MNIST, with RTX 1080Ti GPU support.



It can be seen that for the 2 larger models, there are huge fissures between private max batch size and non-private max batch size, especially for VGG11, the max non-private batch size is almost 15 times more 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 same gap as shown for the two larger model.

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:

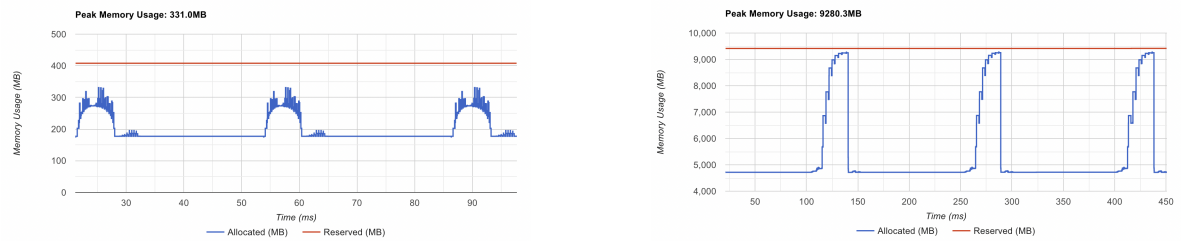


Figure 4: memory usage pattern **Left**: public, **Right**: private

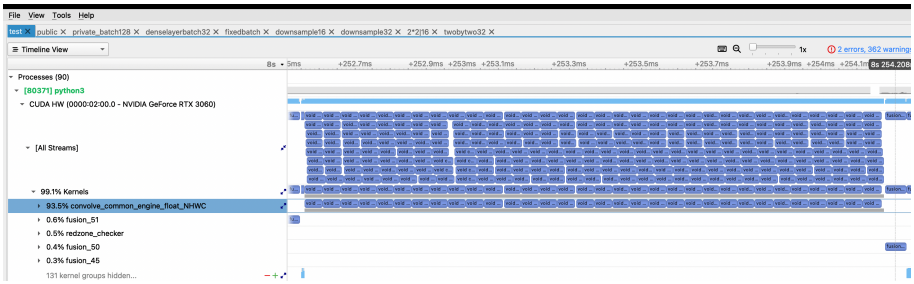
6 Methods We Tried And Further Direction

6.1 Optimize Noising

Based on our previous profiling result using cProfile, it shows that noising is the main 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 optimized, then the gradient calculation part shouldn't raise so much overhead, so we are skeptical that if the implementation of per-sample gradient is fully paralleled in kernel level, which means if the gradient taking steps for each sample happen simultaneously, or there are still imperfections in kernel level parallelism.
The profiling approach we tried is 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 paralleled.
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 [9]. 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.

7 Conclusion

In this paper, we introduce the time and space overhead from introducing differential privacy into stochastic gradient descent training, and present our customized profiling result from PyTorch and JAX. We find that most of the overhead is from per-sample gradient no matter that the model is, we also tried a couple of approaches to speed up the training. In the end, we point out a direction that by writing customized kernel for per-sample gradient, we may be able to solve the problem from another aspect.[1]

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

8 References

[1] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In CCS, pages 1322–1333. ACM, 2015.
[2] Pranav Subramani, Nicholas Vadivelu and Gautam Kamath, https://arxiv.org/pdf/2010.09063.pdf, Oct 2021.
[3] Ian Goodfellow. Efficient per-example gradient computations. arXiv preprint arXiv:1510.01799, 2015.
[4] Gaspar Rochette, Andre Manoel, and Eric W Tramel. Efficient per-example gradient computations in convolutional neural networks. arXiv preprint arXiv:1912.06015, 2019.
[5] Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into backprop. In Proceedings of the 8th International Conference on Learning Representations, ICLR '20, 2020
[6] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Gosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in PyTorch. arXiv preprint arXiv:2109.12298, 2021.
[7] Chris Waites. Pyvacy. https://github.com/ChrisWaites/pyvacy/network, March 2019
[8] Nicolas Papernot, Andrew Galen, and Steven Chien. Tensorflow privacy. https: //github.com/tensorflow/privacy, December 2018
[9] Zhiqi Bu, Jialin Mao, Shiyun Xu, https://arxiv.org/pdf/2205.10683.pdf, Jun 2022
[10] Ian Goodfellow, https://arxiv.org/pdf/1510.01799.pdf, Oct 2015
[11] Xuechen Li, Florian Tramer, Percy Liang, Tatsunori Hashimoto, https://arxiv.org/pdf/2110.05679.pdf, Jul 2022
[12] Martin Abadi, H. Brendan McMahan, Andy Chu, Ilya Mironov, Li Zhang, Ian Goodfellow, Kunal Talwar, https://dl.acm.org/doi/pdf/10.1145/2976749.2978318, Oct 2016