Jaewoo Lee* and Daniel Kifer

Scaling up Differentially Private Deep Learning with Fast Per-Example Gradient Clipping

Abstract: Recent work on Renyi Differential Privacy has shown the feasibility of applying differential privacy to deep learning tasks. Despite their promise, however, differentially private deep networks often lag far behind their non-private counterparts in accuracy, showing the need for more research in model architectures, optimizers, etc. One of the barriers to this expanded research is the training time — often orders of magnitude larger than training non-private networks. The reason for this slowdown is a crucial privacy-related step called "per-example gradient clipping" whose naive implementation undoes the benefits of batch training with GPUs. By analyzing the back-propagation equations we derive new methods for per-example gradient clipping that are compatible with auto-differeniation (e.g., in PyTorch and TensorFlow) and provide better GPU utilization. Our implementation in PyTorch showed significant training speed-ups (by factors of 54x - 94x for training various models with batch sizes of 128). These techniques work for a variety of architectural choices including convolutional layers, recurrent networks, attention, residual blocks, etc.

Keywords: Differential privacy, Deep learning, Gradient clipping, Gradient perturbation

DOI 10.2478/popets-2021-0008
Received 2020-05-31; revised 2020-09-15; accepted 2020-09-16.

1 Introduction

Machine learning models trained on sensitive datasets, such as medical records, emails, and financial transactions have great value to society but also pose risks to individuals who contributed their information to the training data. Even if the model parameters are not shared, black-box access to the models can leak private information [40]. Differential privacy is a promising framework for mitigating such risks because of its strong mathematical guarantees and because of recent advances in differentially private training of predictive models.

Simple models, such as linear regression and logistic regression, which have convenient mathematical structures (e.g., convexity) and relatively few parameters, have been well-studied in the differentially private literature and many privacy preserving training algorithms have been proposed (e.g., [7, 10, 12, 21, 24, 26, 35, 45, 51]). These algorithms are generally fast and accurate compared to their non-private counterparts.

However, state-of-the-art prediction results generally come from deep artificial neural networks with millions of parameters. These models are not convex and hence require different fitting algorithms to ensure privacy [2, 23, 32, 50]. In the non-private case, training is generally accomplished using stochastic gradient descent backed by GPU/TPU hardware accelerators that process multiple training records together in a batch. In the privacy-preserving case, the most generally applicable training algorithm is also a variation of stochastic gradient descent [2]. However, its current implementations (e.g., [31]) are extremely slow because a key step, "per-example gradient clipping",[1] limits the batch-processing capabilities of GPUs/TPUs, resulting in slowdowns of up to two orders of magnitude. This slowdown has a direct impact on differentially-private deep learning research, as it becomes expensive even to experiment with differential privacy and different neural network architectures [6].

In this paper, we show that most of this slowdown can be avoided. By analyzing how backpropagation computes the gradients, we derive some tricks for fast per-example-gradient clipping that are easy to implement and result in speedups of up to 94x over the naive approach. These methods take advantage of auto-differentiation features of standard deep learning packages (such as TensorFlow and PyTorch [33]) and do not require any low-level programming (our code consists

***Corresponding Author: Jaewoo Lee:** University of Georgia, E-mail: jwlee@cs.uga.edu
Daniel Kifer: Penn State University, E-mail: dkifer@cse.psu.edu

[1] Essentially, the gradient contribution of each record in a batch must be normalized first (in a nonlinear way), before the contributions are added together. See Section 3 for details.

of Python wrappers around PyTorch layer objects — e.g., a wrapper for fully connected layers, a wrapper for convolutional layers, etc.).

We note that Goodfellow [17] provided a fast per-example gradient clipping method that only applied to fully connected networks. Our results apply to a wider variety of architectures, including convolutional layers, recurrent networks, attention, residual blocks, etc.

In short, our contributions are as follows.

- We present methods for efficiently computing per-example gradients for different kinds of deep learning models, achieving a speedup of up to 54x to 94x (depending on the model) compared to naive per-example gradient computation on mini-batches of size 128. This allows hardware-accelerated differentially private training of deep learning models to rival the speed of hardware-accelerated non-private training and thus makes differentially private deep learning possible in practical timeframes.
- The proposed methods do not require fundamental changes to GPU parallelization. Instead, they are easy to implement because they take advantage of automatic differentiation capabilities of modern deep learning packages. Our PyTorch wrappers are being prepared for open-source release.
- As an application of the proposed framework, we demonstrate how to train (under Rényi differential privacy) a Transformer encoder block [44], a key component in an architecture that has lead to recent advances in natural language processing.
- We perform extensive experiments and empirically show the effectiveness of approach for differentially private training of various kinds of deep neural network models.

The rest of this paper is organized as follows. In Section 2, we define notations and provide background on differential privacy. Building on these concepts, we describe the per-example gradient clipping problem in Section 3. We then discuss related work in Section 4. We present our proposed methods in Section 5, experimental results in Section 6 and conclusions in Section 7.

2 Preliminaries

In this paper, we use upper-letters (e.g., W) to represent matrices, bold-face lower-case (e.g., $\mathbf{x}$) to represent vectors and non-bold lower-case (e.g., y) to represent scalars. One exception is that D represents a dataset. Tensors of order 3 or higher (i.e., multidimensional arrays that are indexed by 3 or more variables) are represented in calligraphic font (e.g,. $\mathcal{W}$).

We index vectors using square brackets (e.g., $\mathbf{x}[1]$ is the first component of the vector $\mathbf{x}$. For matrices, we use subscripts to identify entries ($W_{i,j}$ is the entry in row i, column j). Similarly, tensors are indexed using subscripts (e.g., $\mathcal{W}_{i,j,k}$ is the entry at row i, column j, depth k). To partially index a matrix or tensor, we use the symbol $*$. That is row i in a matrix W is $W_{i,*}$, column j is $W_{*,j}$. Similarly, for a 4th order tensor $\mathcal{W}$, $\mathcal{W}_{i,j,*,*}$ is a matrix V where $V_{k,\ell} = \mathcal{W}_{i,j,k,\ell}$.

Let $D = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ be a set of n records, where $\mathbf{x}_i \in \mathfrak{X}$ is a feature vector and $y_i \in \mathfrak{Y}$ is a target (value we must learn to predict). We say two datasets D and D' are *neighbors* if D' can be obtained from D by adding or removing one record and write $D{\sim}D'$ to denote this relationship.

2.1 Differential Privacy

Differential privacy is a widely accepted formal privacy definition that requires randomized algorithms (also called *mechanisms* to process data. The intuition behind it is that the addition/deletion of one record should have very little influence on the output distribution.

Definition 1 ((ϵ, δ)-DP [15, 16]). *Given privacy parameters $\epsilon \geq 0$, $\delta \geq 0$, a randomized mechanism (algorithm) $\mathcal{M}$ satisfies (ϵ, δ)-differential privacy if for every set $S \subseteq \mathrm{range}(\mathcal{M})$ and for all pairs of neighboring datasets $D{\sim}D'$,*

$$\Pr[\mathcal{M}(D) \in S] \leq \exp(\epsilon) \Pr[\mathcal{M}(D') \in S] + \delta\,.$$

The probability only depends to the randomness in $\mathcal{M}$.

The cases $\delta = 0$ and $\delta > 0$ are respectively referred to as *pure* and *approximate* differential privacy.

2.2 Rényi Differential Privacy

One of the drawbacks of Definition 1 is that accurately tracking privacy loss from multiple noise-infused accesses to the data is difficult. For this reason, most work on differentially private deep learning uses a variant called Rényi Differential Privacy (RDP) [29] to track privacy leakage of an iterative algorithm and then. At the very end, the RDP parameters are converted to the ϵ, δ parameters of Definition 1. RDP relies on the concept of Rényi divergence:

Definition 2 (Rényi Divergence). *Let P_1 and P_2 be probability distributions over a set Ω and let $\alpha \in (1, \infty)$. Rényi α-divergence $\mathfrak{D}_\alpha$ is defined as: $\mathfrak{D}_\alpha(P_1 \parallel P_2) = \frac{1}{\alpha-1}\log(\mathbb{E}_{x\sim P_2}\left[P_1(x)^\alpha P_2(x)^{-\alpha}\right])$.*

Rényi differential privacy requires two parameters: a moment α and a parameter ϵ that bounds the moment.

Definition 3 ((α, ϵ)-RDP [29]). *Given a privacy parameter $\epsilon \geq 0$ and an $\alpha \in (1, \infty)$, a randomized mechanism $\mathcal{M}$ satisfies (α, ϵ)-Rényi differential privacy (RDP) if for all D_1 and D_2 that differ on the value of one record, $\mathfrak{D}_\alpha(\mathcal{M}(D_1) \parallel \mathcal{M}(D_2)) \leq \epsilon$.*

While the semantics of RDP are still an area of research, its privacy guarantees are currently being interpreted in terms of (ϵ, δ)-differential privacy through the following conversion result [29].

Lemma 1 (Conversion to (ϵ, δ)-DP [29]). *If $\mathcal{M}$ satisfies (α, ϵ)-RDP, it satisfies (ϵ', δ')-differential privacy when $\epsilon' \geq \epsilon + \frac{\log(1/\delta)}{\alpha-1}$ and $\delta' \geq \delta$.*

This result implies that (α, ϵ)-RDP can be converted to (ϵ', δ')-DP for many different choices of ϵ' and δ'. The result can be used in many ways. For example, one may choose a desired ϵ' and set $\delta' = e^{-(\epsilon'-\epsilon)(\alpha-1)}$, in which case (α, ϵ)-RDP provides more protections than differential privacy with those values of ϵ' and δ'. Alternatively, one can pick a δ' and use Lemma 1 to determine the corresponding ϵ'.

Building Blocks. One of the simplest methods of creating an algorithm satisfying RDP is called the Gaussian Mechanism. It relies on a concept called L_2 sensitivity, which measures the largest effect a single record can have on a function. Formally,

Definition 4 (L_2 sensitivity). *Let q be a vector-valued function over datasets. The L_2 sensitivity of q, denoted by $\Delta_2(q)$ is defined as $\Delta_2(q) = \max_{D\sim D'} \|q(D) - q(D')\|_2$, where the max is over all neighboring pairs.*

The Gaussian mechanism for RDP answers a numerical aggregate query q by adding Gaussian noise whose variance depends on the sensitivity of q as follows:

Lemma 2 (Gaussian Mechanism [29]). *Let q be a vector-valued function over datasets. Let $\mathcal{M}$ be a mechanism that releases the random variable $\mathcal{N}(q(D), \sigma^2\mathbf{I}_k)$ and let $\alpha \in (1, \infty)$ and $\epsilon > 0$ be privacy parameters. If $\sigma^2 \geq \alpha\Delta_2^2(q)/(2\epsilon)$, then $\mathcal{M}$ satisfies (α, ϵ)-RDP.*

Composition. More complex algorithms for (α, ϵ)-RDP, such as training deep neural networks, can be created by combining together many applications of simpler mechanisms (such as the Gaussian Mechanism) — each one leaks a controlled amount of private information, and the composition theorem explains how to compute the total leakage.

Lemma 3 (Composition [29]). *Let $\mathcal{M}_1, \dots, \mathcal{M}_k$ be mechanisms such that each $\mathcal{M}_i$ satisfies (α, ϵ_i)-RDP (that is, the α values are all the same but the ϵ values can differ). The mechanism that, on input D, jointly releases the outputs $\mathcal{M}_1(D), \dots, \mathcal{M}_k(D)$ satisfies $(\alpha, \sum_i \epsilon_i)$-RDP.*

In practice, one keeps track of multiple α values. That is, a mechanism $\mathcal{M}_1$ may satisfy (α_1, ϵ_1)-RDP, (α_2, ϵ_2)-RDP and (α_3, ϵ_3)-RDP, while $\mathcal{M}_2$ may satisfy (α_1, ϵ_1')-RDP, (α_2, ϵ_2')-RDP and (α_3, ϵ_3')-RDP. The mechanism that releases both of their outputs would satisfy $(\alpha_1, \epsilon_1 + \epsilon_1')$-RDP and also $(\alpha_2, \epsilon_2 + \epsilon_2')$-RDP and $(\alpha_3, \epsilon_3 + \epsilon_3')$-RDP. When converting to (ϵ, δ)-DP, one applies Lemma 1 to each of these and selects the best ϵ, δ values [2].

Postprocessing Immunity. Another key feature of differential privacy is post-processing immunity. If $\mathcal{M}$ is a mechanism that satisfy (α, ϵ)-RDP (or (ϵ', δ')-DP) and f is any algorithm, then the mechanism which, on input D, releases $f(\mathcal{M}(D))$, satisfies (α, ϵ)-RDP (or (ϵ', δ')-DP) – the privacy parameters do not get worse.

3 The Problem with Per-Example Gradient Clipping

In this section we briefly describe non-private training of neural networks to explain how GPU mini-batch computation is used to speed up training. We then discuss the most common differentially private deep learning training procedure and explain how its direct implementation loses much of these speed benefits (via a step called *gradient clipping*). In Section 5 we then explain how to recover the speedup that was lost with a better gradient clipping algorithm.

3.1 Non-private Mini-batch SGD

A machine learning model $M_{\boldsymbol{\theta}}$ is a parametrized function with parameters $\boldsymbol{\theta}$ (e.g., $\boldsymbol{\theta}$ could be the weights in an artificial neural network). Once the parameters are set, the model can make predictions. The parameters are typically chosen using training data through a process called *empirical risk minimization*: given (1) a dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and (2) a loss function ℓ that quantifies the error between the true target y_i and predicted value $M_\theta(\mathbf{x}_i)$, the goal of empirical risk minimization is to find a value of θ that minimizes

$$\arg\min_{\boldsymbol{\theta}\in\Theta} L(\boldsymbol{\theta}, D) := \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))\,. \tag{1}$$

The function L is called the *objective function.*

When M_θ is a deep neural network, the above problem is typically solved with an iterative first-order algorithm such as stochastic gradient descent (SGD) [9, 36] or its variants.

In each iteration, a set B of τ records is randomly sampled from the data D. This set is called a mini-batch. The objective function is computed over the minibatch: $L(\theta, B) = \frac{1}{\tau}\sum_{\mathbf{x}\in B} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$ and then its gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, B)$ is computed. This gradient is then used to update the parameters $\boldsymbol{\theta}$, either through a vanilla update rule such as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, B)$ (where η is a number called a *learning rate*) or the gradient is used inside a more complicated procedure such as ADAM [25] or RMSProp (see [38] for a survey of alternatives).

The computation of the gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, B)$ is generally the most expensive part of this procedure, but can be thought of as a series of matrix multiplications and element-wise products that can often be performed in parallel. Modern frameworks like TensorFlow [1] and PyTorch [33] use auto-differentiation (e.g., `torch.autograd.grad`) to compute the sum of the gradients over a batch (i.e., $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, B) \equiv \frac{1}{\tau}\sum_{\mathbf{x}\in B} \nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$. Behind the scenes, data records are bulk-loaded onto the GPU to amortize data transfer costs and then the matrix operations take advantage of the parallelism in the GPU.

3.2 Mini-batch Stochastic Gradient Descent with Privacy

In the framework of Abadi et al. [2], adding differential privacy to deep learning requires adding bias and noise into the mini-batch gradient computation. Ideally, one would like to simply add noise to the minibatch gradient $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, B) \equiv \frac{1}{\tau}\sum_{\mathbf{x}\in B} \nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$. To satisfy differential privacy, the noise has to be large enough to mask the effect of any possible record. However, without any further assumptions, a worst-case change to a single record can result in a large change to the mini-batch gradient (potentially large enough to cause floating point computations to result in ∞). The amount of noise necessary to mask such an effect would render all computations useless. Rescaling the inputs (e.g., converting image pixel values from the range [0,255] to [0,1]) would not solve this problem as the millions of weights in a deep network could still result in a large worst-case gradient (this happens even in the non-private setting and is called the exploding gradient problem [18]).

Abadi et al. [2] addressed this problem by clipping each term in the summation to make sure that no term can get large, even in the worst case. The clipping function has a parameter c (called the clipping threshold) and is defined as follows:

$$\text{clip}_c(\mathbf{z}) = \frac{\mathbf{z}}{\max(1, ||\mathbf{z}||_2/c)}$$

If the L_2 norm of a vector is at most c, then $\text{clip}_c(\mathbf{z}) = \mathbf{z}$ (and the L_2 norm of the result is $\leq c$). If the L_2 norm is $> c$ then $\text{clip}_c(\mathbf{z}) = c\frac{\mathbf{z}}{||\mathbf{z}||_2}$, which has a norm equal to c. Hence clip_c always outputs a vector of L_2 norm $\leq c$ that points in the same direction as the input vector.

Thus, the differentially private deep learning framework [2] replaces the mini-batch gradient with $\frac{1}{\tau}\sum_{\mathbf{x}\in B} \text{clip}_c(\nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)))$ and adds Gaussian noise (via the Gaussian mechanism) to this quantity before updating parameters in the network (e.g., the parameters can be updated with this noisy/biased gradient as in vanilla stochastic gradient descent: $\theta \leftarrow \theta - \eta\left(\frac{1}{\tau}\sum_{\mathbf{x}\in B} \text{clip}_c(\nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)))\right)$ or the noisy/biased gradient can be used in more complex rules such as ADAM or RMSProp). Abadi et al. use the Moment Accountant technique [2] to precisely track the privacy protections offered by random sampling (to create the random mini-batches) and the added Gaussian noise.

3.3 The Computational Problem

The efficiency of the differentially private deep learning framework depends on the following question: how does one compute $\frac{1}{\tau}\sum_{\mathbf{x}\in B} \text{clip}_c(\nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)))$? Auto-differentiation software will not do this directly.

One baseline approach (as implemented in TensorFlow Privacy [31]) is to loop through the examples one at a time. For each example $\mathbf{x}_i$ one can ask the auto-differentiator to compute $\nabla_{\boldsymbol{\theta}} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$, then clip it and then at the end, sum up the clipped gradients.

This approach has several drawbacks. First, it loses the parallelism that GPUs can offer when performing matrix computations. Second, it may result in multiple transfers of data to the GPU (i.e., not taking advantage of bulk transfer capabilities).

A related, slightly faster approach is to use the auto-differentiation api to directly ask for multiple gradients. For example in PyTorch, the function `torch.autograd.grad` is normally called with the first parameter equal to the minibatch loss $\frac{1}{\tau}\sum_{\mathbf{x}\in B}\ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$ (in which case it computes the gradient). However, it is also possible to call the function with a vector of losses: $[\ell(y_1, M_{\boldsymbol{\theta}}(\mathbf{x}_1)), \dots, \ell(y_\tau, M_{\boldsymbol{\theta}}(\mathbf{x}_\tau))]$ to obtain the gradient of each one. These gradients can then be clipped and summed together.

In our experiments, this approach is still significantly slower than non-private training. Further significant improvements are possible and are described in Section 5. The main idea is that when deep learning auto-differentiators compute the gradients, they are also computing the derivatives with respect to intermediate variables (e.g., the chain rule). Normally, these intermediate results are not returned but it is possible to ask for them. The per-example gradient norms (i.e. norm of the gradient of each term $\ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$ can be directly computed from these intermediate results. Once the per-example gradient norms are computed, we turn them into weights $\nu_1, \dots, \nu_\tau$ then re-weight the terms in the mini-batch loss: $\frac{1}{\tau}\sum_{\mathbf{x}\in B}\ \nu_i\ \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$. This step ensures that the gradient of each weighted term now has norm at most c. We then ask the audo-differentiator for the gradient of this reweighted loss. The result is exactly equivalent to per-example gradient clipping (but turns out to be much faster than the baseline implementations). Thus, after this reweighted gradient is computed, noise can be added and parameters can be updated as in [2]. We describe the details in Section 5.

4 Related Work

Deep learning for differential privacy was introduced by Skokri and Shmatikov [39] but required enormous values of the privacy parameters (e.g., ϵ values in the hundreds or thousands). The first practical approach, which could train deep networks to reasonable accuracy (on the MNIST and CIFAR datasets) with ϵ values of 10 or less was proposed by Abadi et al. [2] and required the use of gradient clipping and Renyi Differential Privacy [29] (referred to as the Moment Accountant in [2]).

Followup work [3, 4, 8, 13, 28, 42, 50] relied on this training technique. Also [4, 28, 42, 50] investigated different clipping strategies, such as adaptively changing the clipping threshold [4, 42, 50] or clipping the gradient layer by layer [27, 28]. Specifically, given the global clipping threshold c, McMahan et al. [28] clip the gradient of each layer's parameter using the threshold $c/\sqrt{m}$, where m is the total number of layers. In [27], the authors extended the idea of per-layer clipping and proposed a joint clipping strategy which applies different amount of clipping to each group of queries. Since our proposed fast per-example clipping framework is able to compute the per-example gradient norm layer-wise (as well as overall norm), our work can be used to accelerate the previously mentioned training algorithms that experimented with more refined clipping ideas.

There are other approaches to differentially private training of deep networks that avoid gradient clipping and adding noise to gradients. One example is PATE [30, 32] which requires a large private dataset but also a large public dataset (and hence is applicable in fewer scenarios). Gradient clipping in specific models can also be avoided, for example Phan et al. [34] perturb the objective function of auto-encoders while Xie et al. [47] show that it is possible to train a differentially private GAN using weight clipping instead of gradient clipping.

Overall, basing differentially private training algorithms on gradient clipping techniques (e.g., [2]) results in algorithms that are applicable in wider settings. However, despite the popularity of gradient clipping technique in differentially private deep learning, per-example gradient computation for a general neural network was computationally heavy and significantly slowed down training.

In [17], Goodfellow showed that for fully-connected networks per-example gradients can be efficiently computed using auto-differentiation library in deep learning frameworks, such as Tensorflow and PyTorch. A key observation is that in these specific networks, the gradient of loss function L, defined in (1), with respect to the network parameters can be decomposed into the product of intermediate results of the auto-differentiation procedure. Specifically, consider a fully-connected layer with, weight matrix $W \in \mathbb{R}^{m\times n}$ and bias $\mathbf{b} \in \mathbb{R}^m$, whose preactivation $\mathbf{z} \in \mathbb{R}^m$ are computed by $\mathbf{z} = W\mathbf{h} + \mathbf{b}$, where $\mathbf{h} \in \mathbb{R}^n$ is an input vector to the layer (or equivalently, it is the post-activation of the previous layer). A careful analysis using the chain rule reveals that

$$\left\|\frac{\partial L}{\partial W}\right\|_{\mathsf{F}}^2 = \left\|\frac{\partial L}{\partial \mathbf{z}}\right\|^2 \|\mathbf{h}\|^2 .$$

Hence, the norms of per-example gradients can be efficienlty computed (without having to explictly materialize them) if we store $\mathbf{z}$ and $\mathbf{h}$ and compute $\frac{\partial L}{\partial \mathbf{z}}$ using the auto-differentiation library. However, this formula does *not* generalize to other type of neural network layers, e.g., convolutional layer and recurrent layer. We observe that the technique is applicable when the gradient with respect to parameter is expressed as an outer product between the gradient with respect to pre-activation $\frac{\partial L}{\partial \mathbf{z}}$ and the layer input $\mathbf{h}$. That is when

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \mathbf{z}} \otimes \mathbf{h}\,,$$

where $\otimes$ denotes the outer product of two vectors. In this work, we extend the technique to other types of neural networks, derive equations for per-example gradients, and provide a recipe for efficiently computing them and integrating them into differentially private training.

Recently, at the time of writing, Rochette et al. [37] also made an attempt to extend the technique in [17] to convolutional neural networks. While they also analyzed gradients using the chain rule and made observations similar to those in our work, their work differs with ours in both mathematical derivation and implementation. For simplicity, [37] derives the gradient for 1D convolution operation and claim the same result also holds for higher dimensional cases. In our work, we directly show the derivations for 2D convolution (which is most popularly used in practice) using tensors. Another aspect of their technique is that to compute the per-example gradients for 1-D convolutions, they make use of 2-D convolution operations. Extensions of their techniques to per-example gradients for 2-D convolutions would require 3-D convolutions and extensions of their work to 3-D convolutions would not be efficiently supported (for example, due to lack of efficient support of 4-D convolutions in PyTorch). In contrast, to avoid this problem in our implementation, we convert the same operation into one single batch matrix-matrix multiplication, which can be done efficiently on GPUs. In addition, to smoothly integrating our technique into differentially private training, we indirectly clip gradients by assigning weights to loss values, rather than directly manipulating the gradients.

Algorithm 1: Gradient perturbation by reweighting per-example losses

```
Input: dataset $D = \{d_i\}$, model $M_{\boldsymbol{\theta}}$, activation
       function $\phi(\cdot)$, privacy parameters $\epsilon$ and
       $\delta$, number of iterations $T$, mini batch
       size $\tau$, clipping threshold $c$
Use Moment Accountant [2] to determine noise
  variance $\sigma^2$ (based on $m$, $c$, and $T$) that will
  result in $(\epsilon, \delta)$-dp.
for $t = 1, 2, \ldots, T$ do
    Construct a random minibatch $B$ of $\tau$
      records
    $\Gamma = \emptyset\,, \Lambda = \emptyset$
    /* Perform the feed forward step    */
    foreach layer $l$ in $M_{\boldsymbol{\theta}}$ do
        $Z^{(l)} = X^{(l-1)} W^{(l)} + \mathbf{b}^{(l)}$
        $X^{(l)} = \phi(Z^{(l)})$
        $\Gamma = \Gamma \cup \{Z^{(l)}\}$        // pre-activation
        $\Lambda = \Lambda \cup \{X^{(l-1)}\}$        // layer input
    end
    Compute $\frac{\partial L(\boldsymbol{\theta}, B)}{\partial \Gamma}$ via auto-differentiation
                 // same as $\frac{\partial \frac{1}{\tau} \sum_{\mathbf{x} \in B} \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))}{\partial \Gamma}$
    Using $\Lambda$ and $\frac{\partial L(\boldsymbol{\theta}, B)}{\partial \Gamma}$, compute
      $||\nabla \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))||_2$ for $i \in B$ as described
      in Section 5
    $\nu_i \leftarrow \min(1,\, c / \|\nabla \ell(\boldsymbol{\theta}, d_i)\|_2)$
    Use auto-differentiation to compute
      gradient of: $\frac{1}{\tau} \sum_{\mathbf{x} \in B} \nu_i \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$
      // Add Gaussian Noise to the gradient
      (as in [2]) and update parameters.
    $\theta \leftarrow \theta -$
      $\eta \left( N(0, \sigma^2 I) + \frac{1}{\tau} \nabla_{\boldsymbol{\theta}} \sum_{\mathbf{x} \in B}\ \nu_i \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)) \right)$
end
```

5 Faster Deep Learning with Differential Privacy

In this paper, we consider feedforward networks (which include recurrent networks) consisting of layers (e.g., a convolutional layer feeding into a max pooling layer, etc.).

Each layer ℓ has a weight matrix $W^{(\ell)} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ where d_{in} is the number of inputs to the layer and d_{out} is the number of outputs.

Since each example in the mini-batch is being run through the network, we can think of the inputs to the layer as a matrix $X^{(\ell)} \in \mathbb{R}^{\tau \times d_{\text{in}}}$ whose first row (i.e., $X^{(\ell)}_{1,*}$) is the layer's input when the first record of the mini-batch is run through the network and the i^{th} row (i.e., $X^{(\ell)}_{i,*}$) is the layer's input when the i^{th} record of the mini-batch is run through the network.

The pre-activation of the layer is then $X^{(\ell)} W^{(\ell)} + \mathbf{b}$, where $\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_{\text{out}}}$ is the bias parameter of the layer.

The activation function $\phi^{(\ell)}$ of the layer is applied pointwise to the pre-activation to give the post-activation, or output, of the network:

$$X^{(\ell+1)} = \phi^{(\ell)}(Z^{(\ell)}), \quad Z^{(\ell)} = X^{(\ell)}W^{(\ell)} + \mathbf{b}^{(\ell)},$$

In this section we show how to compute $\frac{1}{\tau}\sum_{\mathbf{x}\in B} \text{clip}_c(\nabla_{\boldsymbol{\theta}}\ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)))$. This is the quantity to which Gaussian noise is added and which is then used to update the network parameters during training. Pseudocode for the integration of our procedure into differentially private deep learning in shown in Algorithm 1.

The main idea behind our approach is that $\text{clip}_c(\nabla_\theta \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))) = \nu_i \nabla_\theta \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))$, where

$$\nu_i = \min(1, c/||\nabla_\theta \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i))||_2) \tag{2}$$

If we can compute ν_i for each $\mathbf{x}_i$, then the reweighted loss on the minibatch:

$$\frac{1}{\tau}\sum_{\mathbf{x}\in B} \nu_i \; \ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)) \tag{3}$$

has gradient that equals $\frac{1}{\tau}\sum_{\mathbf{x}\in B} \text{clip}_c(\nabla_{\boldsymbol{\theta}}\ell(y_i, M_{\boldsymbol{\theta}}(\mathbf{x}_i)))$.

Thus we compute ν_i for each i, reweight the loss function, ask the auto-differentiation api to get the gradient, add privacy noise to the gradient, and then update the parameters. The result is exactly the same as per-example gradient clipping, but is much faster.

Noting that the parameters $\boldsymbol{\theta}$ consists of the weight matrix and bias vector of each layer, the L_2 norm of the gradient with respect to $\boldsymbol{\theta}$ is the square root of the sum of squares of the gradients with respect to the $W^{(\ell)}$ and $\mathbf{b}^{(\ell)}$ of each layer.

Thus, in each of the following subsections, we explain how to compute these quantities for each type of layer. All that is needed are quantities $\frac{\partial \text{Loss}}{\partial Z^{(\ell)}}$ (the gradient with respect to pre-activations of Layer ℓ) and $X^{(\ell)}$ (the mini-batch inputs to Layer ℓ).

5.1 Fully-connected Layers

For completeness, we first describe Goodfellow's technique for fully connected layers [17].

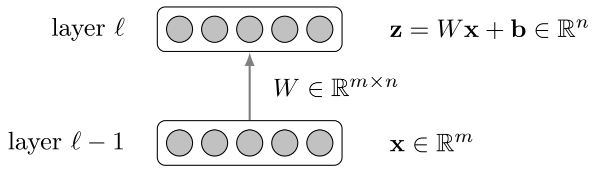

layer ℓ
$\mathbf{z} = W\mathbf{x} + \mathbf{b} \in \mathbb{R}^n$
$W \in \mathbb{R}^{m\times n}$
layer $\ell - 1$
$\mathbf{x} \in \mathbb{R}^m$

Fig. 1. Two fully-connected layers in an MLP

Consider two consecutive fully-connected layers, described in Figure 1, of a multi-layer perceptron (MLP). Let ℓ and $\ell - 1$ denote those two layers. Let $W \in \mathbb{R}^{m\times n}$ be the weight matrix between layers ℓ and $\ell - 1$ and $\mathbf{x} \in \mathbb{R}^n$ be an input to the upper layer (which is also the activation of bottom layer). In the forward phase, the pre-activation $\mathbf{z} \in \mathbb{R}^m$ and activation $\mathbf{a} \in \mathbb{R}^m$ are:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}, \qquad \mathbf{a} = \phi(\mathbf{z}), \tag{4}$$

where ϕ is an activation function applied element-wise and $\mathbf{b} \in \mathbb{R}^m$ is a bias term. By the chain rule, the derivative of L with respect to the entry of W at i^{th} row and j^{th} column is given by

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial W_{i,j}} = \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{z}[k]}\frac{\partial \mathbf{z}[k]}{\partial W_{i,j}}, \tag{5}$$

where we view $\frac{\partial L}{\partial \mathbf{z}}$ and $\frac{\partial \mathbf{z}}{\partial W_{i,j}}$ as matrices of size $1 \times m$ and $m \times 1$, respectively. From (4) we have

$$\begin{aligned}\frac{\partial \mathbf{z}[k]}{\partial W_{i,j}} &= \frac{\partial}{\partial W_{i,j}}\left(\sum_{l=1}^{n} W_{k,l}\mathbf{x}[l] + \mathbf{b}[k]\right) \\ &= \begin{cases}\mathbf{x}[j] & \text{if } k = i, \\ 0 & \text{if } k \neq i.\end{cases}\end{aligned}$$

Plugging the above into (5), we obtain

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \mathbf{z}[i]}\mathbf{x}[j].$$

Combining all together, we see that

$$\begin{aligned}\frac{\partial L}{\partial W} &= \begin{bmatrix} \frac{\partial L}{\partial \mathbf{z}[1]}\mathbf{x}[1] & \frac{\partial L}{\partial \mathbf{z}[1]}\mathbf{x}[2] & \cdots & \frac{\partial L}{\partial \mathbf{z}[1]}\mathbf{x}[n] \\ \frac{\partial L}{\partial \mathbf{z}[2]}\mathbf{x}[1] & \frac{\partial L}{\partial \mathbf{z}[2]}\mathbf{x}[2] & \cdots & \frac{\partial L}{\partial \mathbf{z}[2]}\mathbf{x}[n] \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial \mathbf{z}[m]}\mathbf{x}[1] & \frac{\partial L}{\partial \mathbf{z}[m]}\mathbf{x}[2] & \cdots & \frac{\partial L}{\partial \mathbf{z}[m]}\mathbf{x}[n] \end{bmatrix} \\ &= \frac{\partial L}{\partial \mathbf{z}} \otimes \mathbf{x}, \end{aligned} \tag{6}$$

where $\otimes$ denotes the outer product of two vectors. Equation (6) is the gradient for a single example $\mathbf{x}$. Suppose there are τ examples in the minibatch. Then both $\frac{\partial L}{\partial \mathbf{z}}$ and $\mathbf{x}$ become matrices of size $\tau \times m$ and $\tau \times n$, respectively. To efficiently compute the per-example graidents for τ examples, in our implementation, we reshape $\frac{\partial L}{\partial \mathbf{z}}$ and $\mathbf{x}$ into tensors of size $[\tau, m, 1]$ and $[\tau, 1, n]$, respectively, and perform batch matrix-matrix multiplication[2]. This procedure is described in Algorithm 2. We note that in the pseudocode $\frac{\partial L}{\partial \mathbf{z}_i}$ denotes the gradient for the i^{th} example in the minibatch. Similarly, the gradient

[2] In PyTorch, this is done using `torch.bmm()` function.

```
Algorithm 2: Per-example gradient computation for fully-connected layer
Input: batch of gradients w.r.t. pre-activations
       $Z = [\frac{\partial L}{\partial \mathbf{z}_1}^\intercal, \ldots, \frac{\partial L}{\partial \mathbf{z}_\tau}^\intercal]^\intercal$, batch of layer's
       input $X = [\mathbf{x}_1^\intercal, \ldots, \mathbf{x}_\tau^\intercal]^\intercal$
$\mathcal{Z} \leftarrow$ reshape $Z$ into $[\tau, m, 1]$
$\mathcal{X} \leftarrow$ reshape $X$ into $[\tau, 1, n]$
/* Compute batch matrix-matrix
   multiplication                        */
$\mathcal{G} = \mathsf{bmm}(\mathcal{Z}, \mathcal{X})$
return $\mathcal{G}$
```

of L with respect to the k^{th} entry of bias term $\mathbf{b}[k]$ is

$$\frac{\partial L}{\partial \mathbf{b}[k]} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{b}[k]} = \frac{\partial L}{\partial \mathbf{z}} I_m = \frac{\partial L}{\partial \mathbf{z}}$$

since we have

$$\frac{\partial \mathbf{z}[p]}{\partial \mathbf{b}[k]} = \frac{\partial}{\partial \mathbf{b}[k]} \left(\sum_{l=1}^{m} W_{p,l} \mathbf{x}[l] + \mathbf{b}[p] \right) = \begin{cases} 1 & \text{if } p = k, \\ 0 & \text{if } p \neq k, \end{cases}$$

where I_m denotes an identity matrix of size $m \times m$.

5.2 Convolutional Layers

Suppose we have a convolutional layer with c_{out} kernels of size $\kappa \times \kappa$ [3]. Assume input images have size $s_H \times s_W$ with c_{in} channels. The kernel $\mathcal{W}$ for the layer can be represented by a 4D tensor with dimensions $[c_{\text{out}}, c_{\text{in}}, \kappa, \kappa]$, and the input image $\mathcal{X}$ by a 3D tensor with dimensions $[c_{\text{in}}, s_H, s_W]$. We denote the entry of tensor $\mathcal{X}$ at location (i, j, k) by $\mathcal{X}_{i,j,k}$ and write $\mathcal{X}_{i,j,*}$ to denote the entries of $\mathcal{X}$ whose indices for the first 2 dimensions are fixed to (i, j). $\mathcal{X}_{i:j}$ denotes the entries with indices from i to j.

The pre-activation $\mathcal{Z}$ resulting from performing convolution between $\mathcal{W}$ and $\mathcal{X}$, denoted by $\mathcal{W} * \mathcal{X}$, is expressed as

$$\mathcal{Z}_{l,m,n} = \mathcal{W}_{l,*,*,*} \star \mathcal{X}_{*,m:m+\kappa,n:n+\kappa} + b_l \, , \qquad (7)$$

where $\star$ symbol defines the inner product between two tensors of same order, i.e., $\mathcal{X} \star \mathcal{Y} = \sum_{i,j,k} \mathcal{X}_{i,j,k} \mathcal{Y}_{i,j,k}$. For simplicity, let's fix l and focus on the l^{th} output feature map. See Figure 2 for a graphical depiction of the convolution operation. From (7), we get

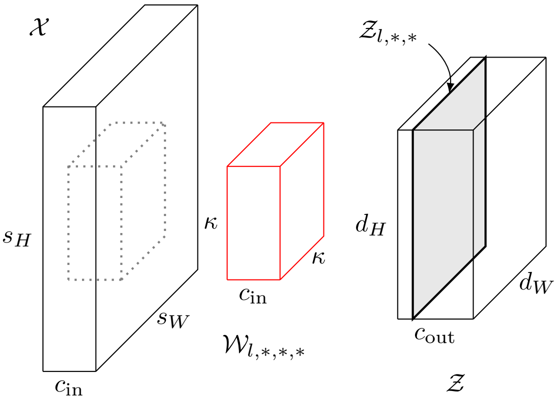


Fig. 2. Convolution between $\mathcal{W}$ and $\mathcal{X}$

$$\frac{\partial \mathcal{Z}_{l,m,n}}{\partial \mathcal{W}_{l,k,i,j}} = \frac{\partial \sum_{p=1}^{\kappa} \sum_{q=1}^{\kappa} \sum_{r=1}^{c_{\text{in}}} \mathcal{W}_{l,r,p,q} \mathcal{X}_{r,m+p-1,n+q-1}}{\partial \mathcal{W}_{l,k,i,j}}$$
$$= \mathcal{X}_{k,m+i-1,n+j-1}$$

and see that the derivative of the l^{th} pre-activation with respect to $\mathcal{W}_{l,k,i,j}$ is given by

$$\frac{\partial \mathcal{Z}_{l,*,*}}{\partial \mathcal{W}_{l,k,i,j}} = \begin{bmatrix} \mathcal{X}_{k,i,j} & \mathcal{X}_{k,i,j+1} & \cdots & \mathcal{X}_{k,i,j+d_W} \\ \mathcal{X}_{k,i+1,j} & \mathcal{X}_{k,i,j+1} & \cdots & \mathcal{X}_{k,i+1,j+d_W} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{X}_{k,i+d_H,j} & \mathcal{X}_{k,i+d_H,j+1} & \cdots & \mathcal{X}_{k,i+d_H,j+d_W} \end{bmatrix},$$

where $d_H = s_H - \kappa$ and $d_W = s_W - \kappa$. Using the chain rule, we get

$$\begin{aligned} \frac{\partial L}{\partial \mathcal{W}_{l,k,i,j}} &= \frac{\partial L}{\partial \mathcal{Z}_{l,*,*}} \frac{\partial \mathcal{Z}_{l,*,*}}{\partial \mathcal{W}_{l,k,i,j}} \\ &= \sum_{m'=1}^{d_H+1} \sum_{n'=1}^{d_W+1} \frac{\partial L}{\partial \mathcal{Z}_{l,m',n'}} \mathcal{X}_{k,i+m'-1,j+n'-1} \qquad (8) \\ &= \frac{\partial L}{\partial \mathcal{Z}_{l,*,*}} \star \mathcal{X}_{k,i:i+d_H,j:j+d_W} \, . \end{aligned}$$

The above equation implies that the gradient $\frac{\partial L}{\partial \mathcal{W}_{l,k,*,*}}$ is obtained by performing convolution between the derivative of L with respect to the pre-activation $\mathcal{Z}_{l,*,*}$ and input image $\mathcal{X}_{k,*,*}$ (without the bias term). That is,

$$\frac{\partial L}{\partial \mathcal{W}_{l,k,*,*}} = \left(\frac{\partial L}{\partial \mathcal{Z}} \right)_{l,*,*} * \mathcal{X}_{k,*,*} \, .$$

As described for the fully-connected layer case, the per-example gradient can be obtained from the derivative $\frac{\partial L}{\partial \mathcal{Z}}$ and layer's input $\mathcal{X}$. The only difference is that we now need to compute the convolution between these two tensors — it was outer product in the fully-connected layer case. To efficiently perform the above convolution

[3] Here we assume the width and height of filter are the same for simplicity. Our result can be generalized to the filters with abitrary size.

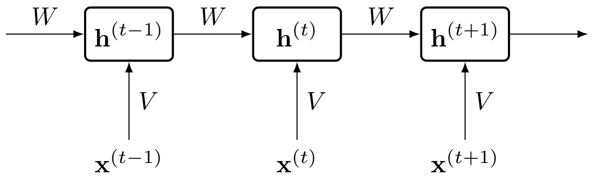


Fig. 3. Recurrent neural network

operation, we convert it into a general matrix-matrix multiplication (GEMM) [11] through vectorizing and reshaping the data and leverage its fast implementation in BLAS library. To this end, we apply im2col [22] transformation on images which converts an image into a matrix where each row corresponds to $\kappa \times \kappa \times C_{\text{in}}$ pixels to which the kernel is applied. See Algorithm 3 for the procedure to get per-example graidents using this operation.

Algorithm 3: Per-example gradient computation for convolutional layer

```
Input: gradient w.r.t. pre-activation $\frac{\partial L}{\partial \mathcal{Z}}$ of
       shape $[\tau, c_{\text{out}}, d_H{+}1, d_W{+}1]$, input
       image $\mathcal{X}$ of shape $[\tau, c_{\text{in}}, s_H, s_W]$
Construct $\mathcal{P} \leftarrow \mathsf{im2col}(\mathcal{X}, [\kappa, \kappa])$
// $\mathcal{P}$ is of shape $[\tau, (d_H{+}1)(d_W{+}1), \kappa^2 c_{\text{in}}]$
$\delta\mathcal{Z} \leftarrow$ reshape $\frac{\partial L}{\partial \mathcal{Z}}$ into $[\tau, c_{\text{out}}, (d_H{+}1)(d_W{+}1)]$
/* Compute batch matrix-matrix
   multiplication                                  */
$\mathcal{G} = \mathsf{bmm}(\delta\mathcal{Z}, \mathcal{P})$
$\mathcal{G} \leftarrow$ reshape $\mathcal{G}$ into $[\tau, c_{\text{out}}, c_{\text{in}}, \kappa, \kappa]$
return $\mathcal{G}$
```

Extensions to 3D convolution. The derivation in (8) readily generalizes to 3D case. Consider a 3D convolution between an input $\mathcal{X}$ of shape $[c_{\text{in}}, d_{\text{in}}, s_H, s_W]$ and a kernel $\mathcal{W}$ of shape $[c_{\text{out}}, c_{\text{in}}, \kappa, \kappa, \kappa]$. The entry at position (o, m, n) of the l^{th} output feature map is given by

$$\mathcal{Z}_{l,o,m,n} = \sum_{p=1}^{\kappa}\sum_{q=1}^{\kappa}\sum_{r=1}^{\kappa}\sum_{c=1}^{c_{\text{in}}} \mathcal{W}_{l,c,r,p,q} \cdot X_{c,o+r,\, m+p-1,n+q-1} \,.$$

From the above, it is easy to see that

$$\frac{\partial \mathcal{Z}_{l,o,m,n}}{\partial \mathcal{W}_{l,c,k,i,j}} = \mathcal{X}_{c,o+k-1,m+i-1,n+j-1}$$

and $\frac{\partial \mathcal{Z}_{l,*,*,*}}{\partial \mathcal{W}_{l,c,k,i,j}}$ is a 4D tensor. As in (8), an application of chain rule yields

$$\begin{aligned}
&\frac{\partial L}{\partial \mathcal{W}_{l,c,k,i,j}} \\
&= \frac{\partial L}{\partial \mathcal{Z}_{l,*,*,*}} \frac{\partial \mathcal{Z}_{l,*,*,*}}{\partial \mathcal{W}_{l,c,k,i,j}} \\
&= \sum_{o'=1}^{d_D+1} \sum_{m'=1}^{d_H+1} \sum_{n'=1}^{d_W+1} \frac{\partial L}{\partial \mathcal{Z}_{l,o',m',n'}} \mathcal{X}_{c,k+o'-1,i+m'-1,j+n'-1} \\
&= \frac{\partial L}{\partial \mathcal{Z}_{l,*,*,*}} \star \mathcal{X}_{c,k:k+d_D,i:i+d_H,j:j+d_W} \,.
\end{aligned}$$

Again, this implies that the gradient of 3D convolution can also be obtained from 3D convolutions.

5.3 Recurrent Layers

We now consider a recurrent layer with weight matrices $W \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{m \times n}$. Let $\mathbf{x}^{(t)} \in \mathbb{R}^n$ and $\mathbf{h}^{(t)} \in \mathbb{R}^m$, for $t = 1, \ldots, T$, denote the input and hidden state vectors at time step t, respectively. As shown in Figure 3, the pre-activation $\mathbf{z}^{(t)} \in \mathbb{R}^m$ at time t is computed by

$$\mathbf{z}^{(t)} = W\mathbf{h}^{(t-1)} + V\mathbf{x}^{(t)} + \mathbf{b} \,, \tag{9}$$

where $\mathbf{h}^{(t-1)} = \phi(\mathbf{z}^{(t-1)})$ and ϕ is an activation function. We first consider the gradient with respect to W, the weight matrix for hidden state vector. By the chain rule, the gradient of L with respect to $W_{i,j}$ is

$$\begin{aligned}
\frac{\partial L}{\partial W_{i,j}} &= \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}^{(t)}} \frac{\partial \mathbf{z}^{(t)}}{\partial W_{i,j}} \\
&= \sum_{t=1}^{T} \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{z}^{(t)}[k]} \frac{\partial \mathbf{z}^{(t)}[k]}{\partial W_{i,j}} \qquad (10) \\
&= \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}^{(t)}[i]} \mathbf{h}^{(t-1)}[j] \qquad (11)
\end{aligned}$$

since we have

$$\begin{aligned}
\frac{\partial \mathbf{z}^{(t)}[p]}{\partial W_{i,j}} &= \frac{\partial}{\partial W_{i,j}} \left(\sum_{k=1}^{m} W_{p,k} \mathbf{h}^{(t-1)}[k] + \mathbf{b}[p] \right) \\
&= \begin{cases} \mathbf{h}^{(t-1)}[j] & \text{if } p = i, \\ 0 & \text{if } p \neq i. \end{cases}
\end{aligned}$$

From (11), we see that

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}^{(t)}} \otimes \mathbf{h}^{(t-1)} \,. \tag{12}$$

Similarly, the gradient with respect to V, weight matrix for input vector, can be obtained as follows:

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}^{(t)}} \otimes \mathbf{x}^{(t)} \;\text{ and }\; \frac{\partial L}{\partial \mathbf{b}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}^{(t)}} \,.$$

Algorithm 4 describes how per-example gradients are computed using Equation (12).

Algorithm 4: Per-example gradient computation for recurrent layer

```
Input: list of gradient w.r.t. pre-activations
       $[\frac{\partial L}{\partial \mathbf{z}^{(1)}}, \ldots, \frac{\partial L}{\partial \mathbf{z}^{(T)}}]$ for $T$ time steps, input
       sequence $[\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}]$
$\Gamma = [\,]$
for $t = 1$ to $T$ do
    $\mathcal{Z} \leftarrow$ shape batch of $\frac{\partial L}{\partial \mathbf{z}^{(t)}}$ into $[\tau, m, 1]$
    $\mathcal{X} \leftarrow$ shape batch of $\mathbf{x}^{(t)}$ into $[\tau, 1, n]$
    $\Gamma$.append(bmm($\mathcal{Z}$, $\mathcal{X}$))
end
$\mathcal{G} \leftarrow$ sum($\Gamma$)
return $\mathcal{G}$
```

5.4 LSTM Layers

The forward phase of an LSTM layer is described by the following pre-activations

$$\begin{bmatrix}\mathbf{z}_f^{(t)}\\ \mathbf{z}_i^{(t)}\\ \mathbf{z}_g^{(t)}\\ \mathbf{z}_o^{(t)}\end{bmatrix} = \begin{bmatrix}W^f\\ W^i\\ W^g\\ W^o\end{bmatrix}\mathbf{h}^{(t-1)} + \begin{bmatrix}V^f\\ V^i\\ V^g\\ V^o\end{bmatrix}\mathbf{x}^{(t)} + \begin{bmatrix}\mathbf{b}^f\\ \mathbf{b}^i\\ \mathbf{b}^g\\ \mathbf{b}^o\end{bmatrix}$$

and 4 gate values $\mathbf{f}^{(t)} = \sigma(\mathbf{z}_f^{(t)})$, $\mathbf{i}^{(t)} = \sigma(\mathbf{z}_i^{(t)})$, $\mathbf{g}^{(t)} = \tanh(\mathbf{z}_g^{(t)})$, and $\mathbf{o}^{(t)} = \sigma(\mathbf{z}_o^{(t)})$, where $\sigma(\cdot)$ is the sigmoid function, $W_\xi \in \mathbb{R}^{m\times m}$ and $V_\xi \in \mathbb{R}^{m\times n}$ for $\xi \in \{f, i, g, o\}$. The above can be simplified by introducing matrices $W \in \mathbb{R}^{4m\times m}$ and $V \in \ R^{4m\times n}$ and a bias $\mathbf{b} \in \mathbb{R}^{4m}$ constructed by stacking weights and biases for all gates:

$$\mathbf{z}^{(t)} = W\mathbf{h}^{(t-1)} + V\mathbf{x}^{(t)} + \mathbf{b}\,.$$

From the above, we see that the gradient of an LSTM layer can be computed in the same way as in a recurrent layer.

5.5 LayerNorm Layers

LayerNorm [5] layer enables a neural network to control the distribution of layer inputs by allowing it to control the mean and variance of inputs across activations (rather than those across minibatch as in batch normalization). It has two parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$. In the forward phase, the LayerNorm at layer ℓ computes the mean and variance of activations from the layer $\ell - 1$:

$$\mu^{(\ell)} = \frac{1}{\kappa}\sum_{i=1}^{\kappa}\mathbf{a}^{(\ell-1)}[i] \text{ and } \sigma^{(\ell)} = \frac{1}{\kappa}\sum_{i=1}^{\kappa}(\mathbf{h}^{(\ell-1)}[i] - \mu^{(\ell)})^2\,.$$

Algorithm 5: Per-example gradient computation for LayerNorm layer

```
Input: batch of gradient w.r.t. pre-activations
       $Z = [\frac{\partial L}{\partial \mathbf{h}_1}^{\intercal}, \ldots, \frac{\partial L}{\partial \mathbf{h}_\tau}^{\intercal}]^{\intercal}$, normalized
       input $H = [\overline{\mathbf{h}}_1^{\intercal}, \ldots, \overline{\mathbf{h}}_\tau^{\intercal}]^{\intercal}$
$\mathcal{G} \leftarrow Z \odot H$
return $\mathcal{G}$
```

It then normalizes the layer inputs by

$$\overline{\mathbf{h}}^{(\ell)}[i] = \frac{1}{\sigma^{(\ell)}}(h^{(\ell-1)}[i] - \mu^{(\ell)})\,, \quad i = 1, \ldots, \kappa\,.$$

Finally, the output of layer is given by

$$\mathbf{h}^{(\ell)} = \boldsymbol{\gamma} \odot \overline{\mathbf{h}}^{(\ell)} + \boldsymbol{\beta}\,,$$

where $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^\kappa$ and $\odot$ denotes the element-wise multiplication. If we view $\mathbf{h}^{(\ell)}$ as the pre-activation of layer, we have

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{\gamma}} &= \frac{\partial L}{\partial \mathbf{h}^{(\ell)}}\frac{\partial \mathbf{h}^{(\ell)}}{\partial \boldsymbol{\gamma}} = \frac{\partial L}{\partial \mathbf{h}^{(\ell)}}\mathrm{diag}(\overline{\mathbf{h}}^{(\ell)}) = \frac{\partial L}{\partial \mathbf{h}^{(\ell)}} \odot \overline{\mathbf{h}}^{(\ell)}\\ \frac{\partial L}{\partial \boldsymbol{\beta}} &= \frac{\partial L}{\partial \mathbf{h}^{(\ell)}}\frac{\partial \mathbf{h}^{(\ell)}}{\partial \boldsymbol{\beta}} = \frac{\partial L}{\partial \mathbf{h}^{(\ell)}}I_k = \frac{\partial L}{\partial \mathbf{h}^{(\ell)}}\end{aligned}$$

since we have

$$\begin{aligned}\frac{\partial \mathbf{h}^{(\ell)}[p]}{\partial \boldsymbol{\gamma}[i]} &= \frac{\partial}{\partial \boldsymbol{\gamma}[i]}\left(\boldsymbol{\gamma}[p]\overline{\mathbf{h}}^{(\ell)}[p] + \boldsymbol{\beta}[p]\right)\\ &= \begin{cases}\overline{\mathbf{h}}^{(\ell)}[i] & \text{if } p = i,\\ 0 & \text{if } p \neq i,\end{cases}\end{aligned}$$

and

$$\frac{\partial \mathbf{h}^{(\ell)}[p]}{\partial \boldsymbol{\beta}[i]} = \frac{\partial}{\partial \boldsymbol{\gamma}[i]}\left(\boldsymbol{\gamma}[p]\overline{\mathbf{h}}^{(\ell)}[p] + \boldsymbol{\beta}[p]\right) = \begin{cases}1 & \text{if } p = i,\\ 0 & \text{if } p \neq i.\end{cases}$$

As shown in Algorithm 5, the per-example gradient for LayerNorm layer over a minibatch can be obtained by simple element-wise product of two matrices.

5.6 Multi-head Attention Layers

Multi-head attention mechanism is a core component of TRANSFORMER network [14, 44, 48], the state-of-the-art model for neural language translation (NLT).

Let $X = (\mathbf{x}_1^{\intercal}, \mathbf{x}_2^{\intercal}, \ldots, \mathbf{x}_s^{\intercal})^{\intercal}$ be an input sequence of encoded vectors $\mathbf{x}_i \in \mathbb{R}^{d_m}$, and consider a multi-head attention layer with h attention heads in the l^{th} layer of a TRANSFORMER network. The architecture of a transformer network with a single encoder

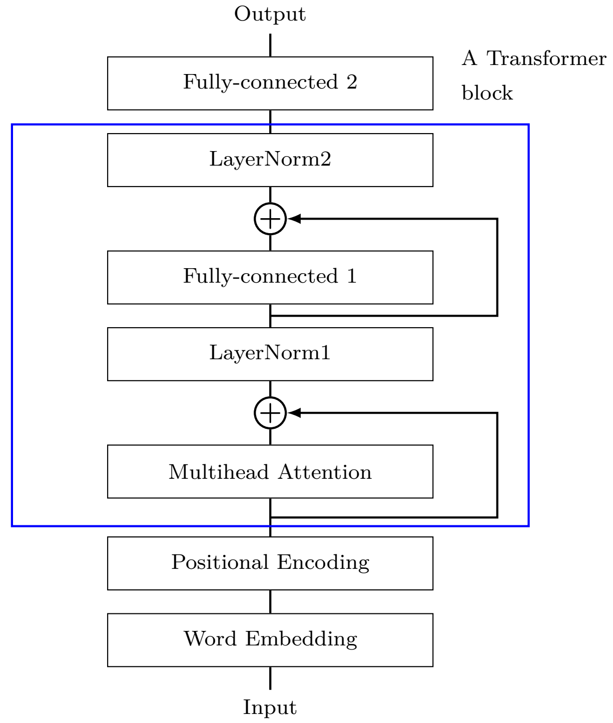


Fig. 4. A transformer network with a single encoder block.

block is described in Figure 4. The layer takes a tuple $(Q^{(\ell-1)}, K^{(\ell-1)}, V^{(\ell-1)})$ of query Q, key K, and value V from the layer $\ell - 1$ as input. Note that $(Q^{(0)}, K^{(0)}, V^{(0)}) = (X, X, X)$. It starts by applying linear transformations on the inputs. This is done by multiplying them with weight matrices $W^Q, W^K, W^V \in \mathbb{R}^{d_m \times d_m}$:

$$\begin{aligned} Q^{(\ell)} &= Q^{(\ell-1)}(W^Q)^\intercal\,, \\ K^{(\ell)} &= K^{(\ell-1)}(W^K)^\intercal\,, \\ V^{(\ell)} &= V^{(\ell-1)}(W^V)^\intercal\,. \end{aligned}$$

The attention weights are computed by the scaled dot product between Q and K. The attention values are weighted sums of values V.

$$\begin{aligned} A^{(\ell)} &= \frac{1}{\sqrt{d_k}}\text{softmax}\left(Q^{(\ell)}(K^{(\ell)})^\intercal\right)\,, \\ H^{(\ell)} &= A^{(\ell)}V^{(\ell)}\,, \end{aligned}$$

where $d_m = h \times d_k$. Finally, the output of layer Y is obtained by applying a linear transformation on the attention values:

$$Y^{(\ell)} = H^{(\ell)}(W^O)^\intercal\,,$$

where $W^O \in \mathbb{R}^{d_m \times d_m}$. The gradient of L with respect to W^Q is

$$\frac{\partial L}{\partial W^Q} = \frac{\partial L}{\partial Q^{(\ell)}}\frac{\partial Q^{(\ell)}}{\partial W^Q}\,.$$

From

$$\begin{aligned} \frac{\partial Q^{(\ell)}_{m,n}}{\partial W_{i,j}} &= \frac{\partial}{\partial W_{i,j}}\left(\sum_{k=1}^{d_m} Q^{(\ell-1)}_{m,k} W_{k,n}\right) \\ &= \begin{cases} Q^{(\ell)}_{m,j} & \text{if } n = i, \\ 0 & \text{if } n \neq i, \end{cases}\,, \end{aligned}$$

we get

$$\frac{\partial Q^{(\ell)}}{\partial W_{i,j}} = \begin{bmatrix} 0 & \cdots & Q^{(\ell-1)}_{1,j} & \cdots & 0 \\ 0 & \cdots & Q^{(\ell-1)}_{2,j} & \cdots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \cdots & Q^{(\ell-1)}_{s,j} & \cdots & 0 \end{bmatrix}\,,$$

where we only have non-zero entries at the i^{th} column. From the above, we have

$$\frac{\partial L}{\partial W_{i,j}} = \sum_{k=1}^{s}\frac{\partial L}{\partial Q^{(\ell)}_{k,i}} Q^{(\ell-1)}_{k,j} = \left\langle \frac{\partial L}{\partial Q^{(\ell)}_{*,i}}, Q^{(\ell-1)}_{*,j} \right\rangle\,.$$

In other words, the entry of $\frac{\partial L}{\partial W}$ at location (i, j) is obtained by taking inner product between the i^{th} column of $\frac{\partial L}{\partial Q^{(\ell)}}$ and the j^{th} column of $Q^{(\ell-1)}$. Combining all together, we conclude that

$$\frac{\partial L}{\partial W^Q} = \left(\frac{\partial L}{\partial Q^{(\ell)}}\right)^\intercal Q^{(\ell-1)}\,.$$

Similarly, we can compute the gradients with respect to other parameters:

$$\begin{aligned} \frac{\partial L}{\partial W^K} &= \left(\frac{\partial L}{\partial K^{(\ell)}}\right)^\intercal K^{(\ell-1)}\,, \\ \frac{\partial L}{\partial W^V} &= \left(\frac{\partial L}{\partial V^{(\ell)}}\right)^\intercal V^{(\ell-1)}\,, \\ \frac{\partial L}{\partial W^O} &= \left(\frac{\partial L}{\partial Y^{(\ell)}}\right)^\intercal H^{(\ell)}\,. \end{aligned}$$

5.7 Other Layer Types

There are many types of layers that have no parameters at all. Some common examples include max-pooling layers (which divide the layer input into patches and output the maximum of each patch), softmax layers (which take a vector $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ and output a vector $\left[\frac{e^{\mathbf{x}[1]}}{\sum_i e^{\mathbf{x}[i]}}, \cdots, \frac{e^{\mathbf{x}[d_{\text{in}}]}}{\sum_i e^{\mathbf{x}[i]}}\right]$). These layers do not outwardly affect our approach – they are automatically accounted for when we ask the auto-differentiation software to compute $\frac{\partial \text{Loss}}{\partial Z^{(\ell)}}$ for layers ℓ below them.

Similarly, skip-connections, which are used in residual blocks [19] also do not outwardly affect our approach.

5.8 Implementation

We implemented the fast per-example gradient clipping technique, described in Section 5, using PyTorch. We encapsulated the per-example gradient norm computation functionality into python wrapper classes for PyTorch's built-in network layers, e.g., Linear, Conv2D, RNN, and so on. This modular implementation allows users to incorporate the gradient clipping functionality into their existing neural network models by simply replacing their layers with our wrapper classes. Each layer wrapper class maintains references to two tensors: pre-activations Z and input X to the layer. After the feed-forward step, it computes $\partial L/\partial Z$, the gradient with respect to Z, using `autograd` package and combines it with X to derive per-example gradients.

6 Experiments

6.1 Experimental Setup

To evaluate the efficiency of the proposed framework, we compare the performance of our per-example loss reweighting algorithm to those of two other algorithms, namely Non-private and nxBP, on different types of neural network models. Non-private algorithm takes a minibatch of examples and performs the forward and backward propagation steps only once as in standard training process. nxBP is the baseline differentially private deep learning algorithm that computes per-example gradient clipping using the naive method from Section 3: it uses auto-differentiation to sequentially obtain the gradient for each record, clips it, and then adds the clipped gradients together. multiLoss is an improved version of the naive approach. As described in Section 3, it asks the auto-differentiator to get the gradients for all examples at once (e.g., it calls `torch.autograd.grad` with first parameter equal to the vector of losses across minibatch records) and then clips and adds them together. Our algorithm, ReweightGP, performs back-propagation twice, once for computing per-example gradient norms (as explained in Section 5) to determine the weights for individual loss functions and the other for computing the batch gradient of weighted loss function.

We note that accuracy comparisons among the differentially private algorithms are irrelevant, as they all produce the same clipped gradients – the only difference among them is speed.

We have implemented our algorithm using PyTorch [33] framework. We used a differentially private version of Adam optimizer, which is the same with the non-private Adam [25] except it injects Gaussian noise with scale σ to gradients. In our experiments, we set the default value for the clipping threshold C to be 1 and used the default value of $\sigma = 0.05$. For all experiments, we set the step size of Adam optimizer to 0.001, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. At each epoch, we randomly shuffle the dataset and partition the data into non-overlapping chunks of size $|B|$. All the experiments were conducted on a machine with Intel Xeon E5-2660 CPU and NVIDIA GeForce 1080 TI GPU.

6.1.1 Models

We tested the effectiveness of our framework on the following 5 different neural network models for classification. All models apply softmax function to the output layers and use the cross entropy loss.

- MLP (Multi-layer Perceptron): this is a simple neural network with two hidden layers. The first layer contains 128 and the second layer 256 units. We used sigmoid function as our default activation function.
- CNN (Convolution Neural Network): the network consists of 2 convolutional layers, each of which followed by a 2 × 2 max pooling layer with stride of 2, and one fully connected layer with 128 hidden units. The first convolutional layer has 20 kernels of size 5×5 with stride 1, and the second layer 50 kernels of size 5 × 5 with stride 1. We didn't use zero-paddings.
- RNN (Recurrent Neural Network): this network was constructed by adding a fully connected layer on top of one vanilla recurrent layer with 128 hidden units. tanh was used as an activation function.
- LSTM (Long Short-term Memory): similar to RNN, there is one LSTM layer with 128 hidden units followed by a fully connected layer for classification.
- Transformer: the network contains a word embedding layer, positional encoding layer, a transformer encoder block, and a fully connected layer. Figure 4 describes the architecture of the Transformer network used in our experiments.

6.1.2 Datasets and Tasks

We used the following five publicly available datasets in our experiments.

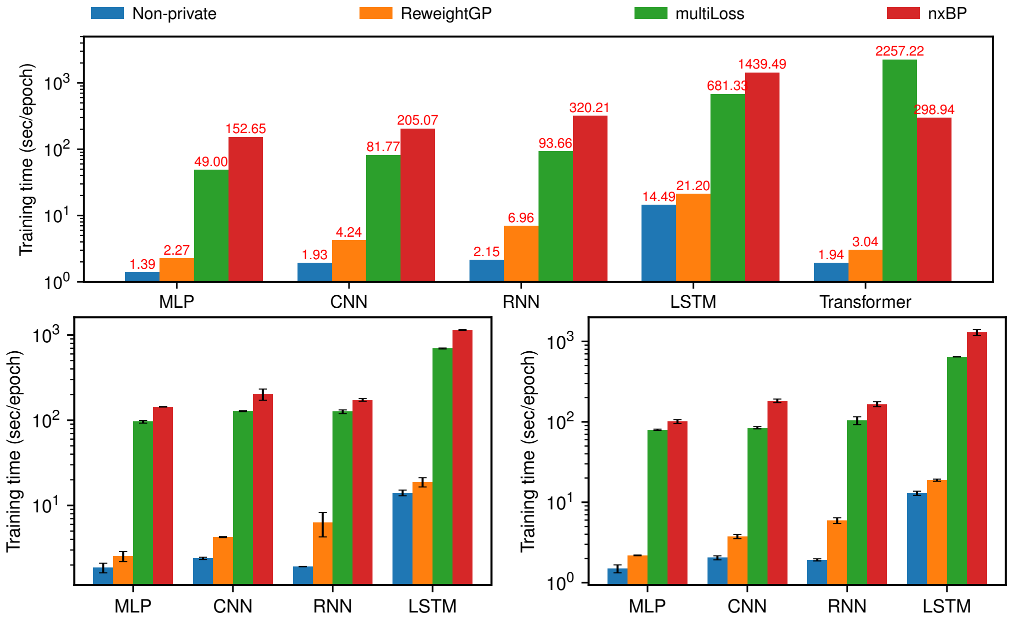


Fig. 5. Comparison of performance by varying architectures (Top: MNIST, Bottom-left: FMNIST, Bottom-right: CIFAR10, Transformer is trained on IMDB)

1. **MNIST** is a grayscale, image dataset of handwritten digits, consisting of 60,000 training and 10,000 test examples. Each image has 28 × 28 pixels, and there are 10 classes (one for each digit). We trained MLP, CNN, RNN, and LSTM networks for classfication. For RNN and LSTM, we construct a sequence by considering the i^{th} row of an image as an input vector for the time step i. In other words, we view an image as a sequence of rows.
2. **FMNIST** (Fashion-MNIST) is a dataset of fashion article images designed to replace MNIST dataset. It also contains 70,000 grayscale images of size 28×28 (60,000 for training and 10,000 for testing).
3. **CIFAR10** is an image dataset for object classification. It consists of 50,000 training examples of 32×32 RGB images. There are 10 classes, and each class has 5,000 images.
4. **IMDB** is a movie review dataset for binary sentiment analysis. We trained the Transformer network on this dataset using 50% of examples. The other 50% of examples were used for testing. For word embedding, rather than training from scratch, we leveraged GloVe embedding vectors of 200 dimensions, pretrained on 6 billions of tokens.
5. **LSUN** [49] is a large-scale scene understanding dataset, having over 59 million RGB images of size at least 256× 256, and 10 different scene categories.

6.2 Small Image Performance

We first show improvements for each architecture on the smaller image datasets (MNIST, FMNIST, CIFAR10). These datasets are not appropriate for Transformer, so we use IMDB for this architecture. Figure 5 compares the performance of the different gradient clipping computation methods on 5 different neural network models in terms of training time per epoch. For this experiment, the minibatch size $|B|$ was fixed to 32, and the models were trained for 100 epochs. As shown in the Figure 5, the proposed ReweightGP algorithm significantly reduces the training time on all 5 different architectures. Notice that values on y-axis are in log scale. It is worth noting that the training of LSTM network takes significantly longer than that for other networks because the per-gradient computation must access each layer's pre-activations and input tensor. This prevents us from using highly optimized fast implementation of LSTM such as NVIDIA's cuDNN LSTM. For RNN, this limitation can be avoided as one can derive the gradient of loss function with respect to pre-activations from the gradient with respect to activations using the chain rule.

6.3 Impact of Different Batch Size

Figure 6 shows the impact of different batch sizes on the per-epoch training time. For this experiment, we

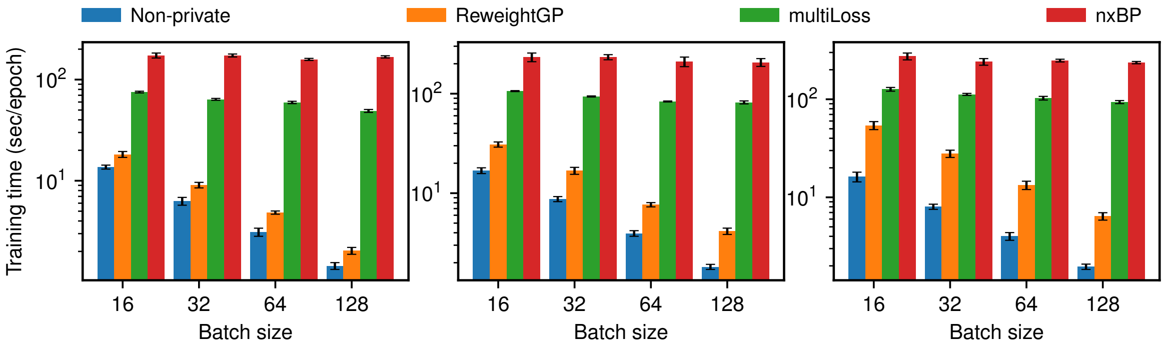


Fig. 6. Execution time by varying batch size (Left: MLP, Middle: CNN, Right: RNN)

trained the MLP, CNN, and RNN models described in Section 6.1.1 on MNIST dataset by varying the batch size. The batch sizes used for training are 16, 32, 64, and 128. An interesting observation is that for Non-private and ReweightGP per-epoch training time decreases as the batch size increases, while that for nxBP remains constant regardless of batch size. This is because that both Non-private and ReweightGP can take advantage of more parallelism due to the use of larger batch. On the other hand, in nxBP computationally heavy error backpropagation happens for each training example (even if an entire batch is stored in the gpu).

6.4 Impact of Network Depth

Before experimenting with larger and more complex architectures, we first provide network depth results for smaller architectures, as small networks are most commonly used with differential privacy [2]. We trained multiple MLP models on three datasets (MNIST, FMNIST, and CIFAR10) by using different numbers of hiddne layers: 2, 4, 6, and 8. The batch size $|B|$ is fixed to 128. As shown in Figure 7, ReweightGP algorithm significantly outperforms the naive nxBP algorithm on all three datasets. Especially on FMNIST dataset with 2 hidden layers, the proposed algorithm showed 94x speed-up over the naive nxBP algorithm.

6.5 ResNet and VGG Networks

We now evaluate the performance on deeper architectures with millions of parameters: ResNet [19] and VGG networks [41]. For this evaluation, we froze the batchnorm parameters at values taken from pre-trained models (since batch-norm parameters do not have per-example gradients). [4] Due to the large memory space requirement, mini-batches of size 20 are used for this experiment. Results on the LSUN dataset are shown in Figure 8. mutiLoss had out-of-memory errors on VGG networks and resnet101 for large images. We still see that ReweightGP consistently outperforms other gradient clipping algorithms (nxBP, multiLoss). The improvement is significant for images of (rescaled) size 64x64 and diminishes for size 256x256.

6.6 Image Size

Noting that image size played a key role in reducing the speedup, we investigate this further in Figure 9 using ResNet 18 with batch size 32 and image sizes ranging from 32x32 to 256x256. This causes quadratic growth in the width of the network (multiplying each dimension by c results in c^2 as many pixels) and we see that the advantage over the naive method decreases due to the extra computation per layer that ReweightGP uses.

6.7 Memory

Due to caching, it is difficult to obtain an accurate estimate of GPU memory requirements. As an alternative, we consider the largest batch size a method can support before running out of memory. For this experiment, we used ResNet 101 with 256x256 input images and varied the batch sizes. The non-private method first failed at batch size 48, ReweightGP at 36, and multiLoss at 18. nxBP operates on one example at a time (even when an entire batch is stored in the GPU). Thus we estimate

[4] In practice, other types of normalizations could also be used, such as LayerNorm [5] (Section 5.5), group norm [46], and instance norm [43].

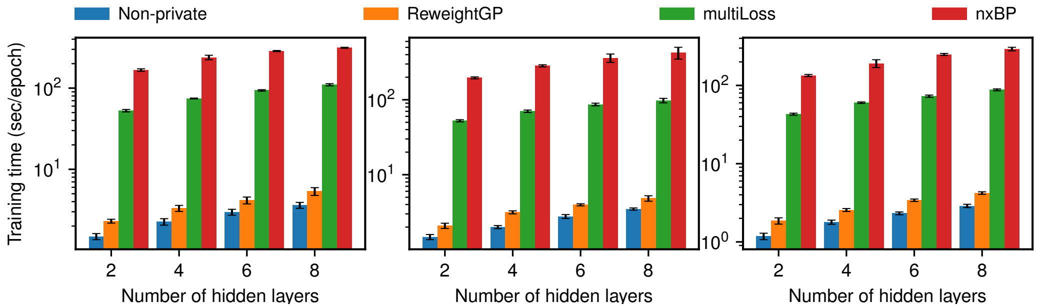
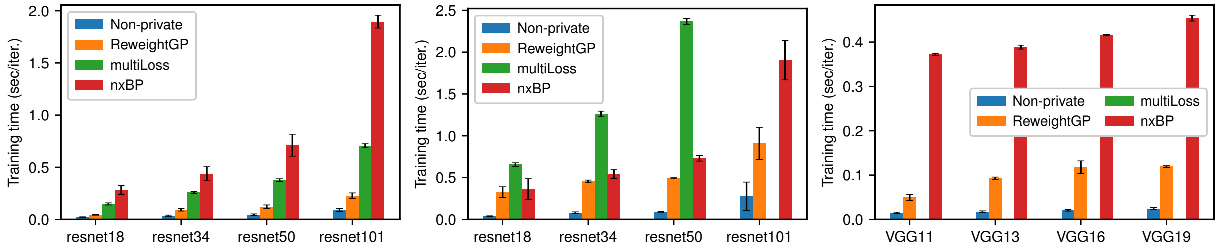


Fig. 7. Comparison of performance by varying number of hidden layers (Left: MNIST, Middle: FMNIST, Right: CIFAR10)



(a) ResNet (Image size: 64×64)

(b) ResNet (Image size: 256×256)

(c) VGG (Image size: 64× 64)

Fig. 8. Performance evaluation on ResNet and VGG networks. Bar for multiLoss is missing when it runs out of memory.

the GPU memory overhead of ReweightGP compared to nonprivate to be up to $(48 - 36)/48 \approx 25\%$ for large images. At the lower end, ReweightGP with ResNet 18 with 32x32 images ran with batch size of 500 without any problems. Note nxBP under-utilizes GPU memory and parallelism (backpropagating through one example at a time). Thus, in practice, the memory overhead is manageable (i.e., allows for relatively large batch sizes) and buys us significant improvements in running time (taking better advantage of GPU parallelism).

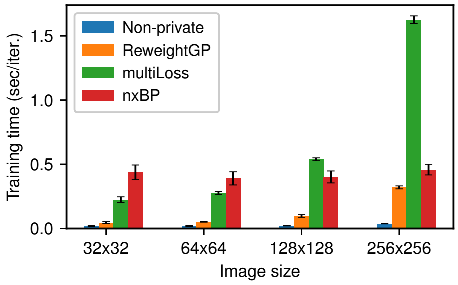


Fig. 9. Processing time by image resolution

6.8 Limitations

Overall, the experiments have shown that our proposed ReweightGP method outperforms the other methods nxBP and MultiLoss (which is often unreliable). ReweightGP requires more memory and computation per layer than nxBP. As a result, its advantage starts to decline with increased image sizes as this causes a quadratic scaling in the width of the network and consequently in the computations of ReweightGP. For very high resolution images, it may be preferable to use nxBP.

Second, some highly optimized versions of LSTM, such as the ones that use the CuDNN LSTM routines do not expose the internal gate values, so that we cannot obtain the appropriate gradients. However, less optimized versions of LSTM can be implemented in PyTorch/TensorFlow and benefit from our approach.

7 Conclusions

We presented a general framework for fast per-example gradient clipping which can be used to improve training speed under differential privacy. Prior work underutilized GPU parallelism, leading to slow training times. Our empirical evaluation showed a significant reduction in training time of differentially private models.

Per-example gradient clipping is not compatible with Batch Norm [20], but other layer normalization methods can be used instead [5, 43, 46].

Acknowledgements

This work was supported by NSF Awards CNS-1931686 and CNS-1943046.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[3] N. C. Abay, Y. Zhou, M. Kantarcioglu, B. M. Thuraisingham, and L. Sweeney. Privacy preserving synthetic data release using deep learning. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2018, Dublin, Ireland, September 10-14, 2018, Proceedings, Part I*, pages 510–526, 2018.

[4] G. Acs, L. Melis, C. Castelluccia, and E. D. Cristofaro. Differentially private mixture of generative neural networks. In *ICDM*, 2017.

[5] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[6] E. Bagdasaryan and V. Shmatikov. Differential privacy has disparate impact on model accuracy. *CoRR*, abs/1905.12101, 2019.

[7] R. Bassily, A. Smith, and A. Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, FOCS '14, pages 464–473, Washington, DC, USA, 2014. IEEE Computer Society.

[8] B. K. Beaulieu-Jones, Z. S. Wu, C. Williams, R. Lee, S. P. Bhavnani, J. B. Byrd, and C. S. Greene. Privacy-preserving generative deep neural networks support clinical data sharing. *bioRxiv*, 2018.

[9] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[10] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate. Differentially private empirical risk minimization. *Journal of Machine Learning Research*, 12(Mar):1069–1109, 2011.

[11] K. Chellapilla, S. Puri, and P. Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[12] C. Chen, J. Lee, and D. Kifer. Renyi differentially private erm for smooth objectives. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2037–2046, 2019.

[13] Q. Chen, C. Xiang, M. Xue, B. Li, N. Borisov, D. Kaafar, and H. Zhu. Differentially private data generative models. https://arxiv.org/pdf/1812.02274.pdf, 2018.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[15] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

[16] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[17] I. Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

[18] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[19] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[20] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[21] R. Iyengar, J. P. Near, D. Song, O. Thakkar, A. Thakurta, and L. Wang. Towards practical differentially private convex optimization. In *Towards Practical Differentially Private Convex Optimization*, page 0. IEEE.

[22] Y. Jia. *Learning semantic image representations at a large scale*. PhD thesis, UC Berkeley, 2014.

[23] J. Jordon, J. Yoon, and M. van der Schaar. Pate-gan: Generating synthetic data with differential privacy guarantees. In *ICLR*, 2019.

[24] D. Kifer, A. Smith, and A. Thakurta. Private convex empirical risk minimization and high-dimensional regression. In *Conference on Learning Theory*, pages 25–1, 2012.

[25] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[26] J. Lee and D. Kifer. Concentrated differentially private gradient descent with adaptive per-iteration privacy budget. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[27] H. B. McMahan, G. Andrew, U. Erlingsson, S. Chien, I. Mironov, N. Papernot, and P. Kairouz. A general approach to adding differential privacy to iterative training procedures. *arXiv preprint arXiv:1812.06210*, 2018.

[28] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. Learning differentially private recurrent language models. In *International Conference on Learning Representations*, 2018.

[29] I. Mironov. Renyi differential privacy. In *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*, pages 263–275. IEEE, 2017.

[30] N. Papernot, M. Abadi, Úlfar Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *Proceedings of the International Conference on Learning Representations*, 2017.

[31] N. Papernot, S. Chien, C. C. Choo, G. M. Andrew, and I. Mironov. TensorFlow Privacy.

[32] N. Papernot, S. Song, I. Mironov, A. Raghunathan, K. Talwar, and Úlfar Erlingsson. Scalable private learning with pate. In *International Conference on Learning Representations (ICLR)*, 2018.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

[34] N. Phan, Y. Wang, X. Wu, and D. Dou. Differential privacy preservation for deep auto-encoders: an application of human behavior prediction. In *AAAI*, 2016.

[35] M. Reimherr and J. Awan. KNG: the k-norm gradient mechanism. In *NeurIPS*, 2019.

[36] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[37] G. Rochette, A. Manoel, and E. W. Tramel. Efficient per-example gradient computations in convolutional neural networks. *ArXiv*, abs/1912.06015, 2019.

[38] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[39] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[40] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[41] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[42] O. Thakkar, G. Andrew, and H. B. McMahan. Differentially private learning with adaptive clipping. *CoRR*, abs/1905.03871, 2019.

[43] D. Ulyanov, A. Vedaldi, and V. S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[45] D. Wang, M. Ye, and J. Xu. Differentially private empirical risk minimization revisited: Faster and more general. In *Advances in Neural Information Processing Systems 30*, pages 2719–2728. Curran Associates, Inc., 2017.

[46] Y. Wu and K. He. Group normalization. In *ECCV*, 2018.

[47] L. Xie, K. Lin, S. Wang, F. Wang, and J. Zhou. Differentially private generative adversarial network, 2018.

[48] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[49] F. Yu, Y. Zhang, S. Song, A. Seff, and J. Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *ArXiv*, abs/1506.03365, 2015.

[50] L. Yu, L. Liu, C. Pu, M. E. Gursoy, and S. Truex. Differentially private model publishing for deep learning. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 332–349, 2019.

[51] J. Zhang, K. Zheng, W. Mou, and L. Wang. Efficient private erm for smooth objectives. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3922–3928. AAAI Press, 2017.