Matrix Sketching for Secure Collaborative Machine Learning

Mengjiao Zhang [1] **Shusen Wang** [1]

Abstract

Collaborative learning allows participants to jointly train a model without data sharing. To update the model parameters, the central server broadcasts model parameters to the clients, and the clients send updating directions such as gradients to the server. While data do not leave a client device, the communicated gradients and parameters will leak a client's privacy. Attacks that infer clients' privacy from gradients and parameters have been developed by prior work. Simple defenses such as dropout and differential privacy either fail to defend the attacks or seriously hurt test accuracy. We propose a practical defense which we call Double-Blind Collaborative Learning (DBCL). The high-level idea is to apply random matrix sketching to the parameters (aka weights) and re-generate random sketching after each iteration. DBCL prevents clients from conducting gradient-based privacy inferences which are the most effective attacks. DBCL works because from the attacker's perspective, sketching is effectively random noise that outweighs the signal. Notably, DBCL does not much increase computation and communication costs and does not hurt test accuracy at all.

1. Introduction

Collaborative learning allows multiple parties to jointly train a model using their private data but without sharing the data. Collaborative learning is motivated by real-world applications, for example, training a model using but without collecting mobile user's data.

Distributed stochastic gradient descent (SGD) is perhaps the simplest approach to collaborative learning. Specifically, the central server broadcasts model parameters to the clients, each client uses a batch of local data to compute a stochastic gradient, and the server aggregates the stochastic gradients and updates the model parameters. Based on distributed SGD, communication-efficient algorithms such as federated averaging (`FedAvg`) (McMahan et al., 2017) and `FedProx` (Sahu et al., 2019) have been developed and analyzed (Li et al., 2019b; Stich, 2018; Wang & Joshi, 2018; Yu et al., 2019; Zhou & Cong, 2017).

Collaborative learning seemingly protects clients' privacy. Unfortunately, this has been demonstrated not true by recent studies (Hitaj et al., 2017; Melis et al., 2019; Zhu et al., 2019). Even if a client's data do not leave his device, important properties of his data can be disclosed from the model parameters and gradients. To infer other clients' data, the attacker needs only to control one client device and access the model parameters in every iteration; the attacker does not have to take control of the server (Hitaj et al., 2017; Melis et al., 2019; Zhu et al., 2019).

The reason why the attacks work is that model parameters and gradients carry important information about the training data (Ateniese et al., 2015; Fredrikson et al., 2015). Hitaj et al. (2017) used the jointly learned model as a discriminator for training a generator which generates other clients' data. Melis et al. (2019) used gradient for inferring other clients' data properties. Zhu et al. (2019) used both model parameters and gradients for recovering other clients' data. Judging from published empirical studies, the gradient-based attacks (Melis et al., 2019; Zhu et al., 2019) are more effective than the parameter-based attack (Hitaj et al., 2017). Our goal is to defend the gradient-based attacks.

Simple defenses, e.g., differential privacy (Dwork, 2011) and dropout (Srivastava et al., 2014), have been demonstrated not working well by (Hitaj et al., 2017; Melis et al., 2019). While differential privacy (Dwork, 2011), i.e., adding noise to model parameters or gradients, works if the noise is strong, the noise inevitably hurts the accuracy and may even stop the collaborative learning from making progress (Hitaj et al., 2017). If the noise is not strong enough, clients' privacy will leak. Dropout training (Srivastava et al., 2014) randomly masks a fraction of the parameters, making the clients have access to only part of the parameters in each iteration. However, knowing part of the parameters is sufficient for conducting the attacks.

[1] Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030. Correspondence to: Mengjiao Zhang <mzhang49@stevens.edu>, Shusen Wang <shusen.wang@stevens.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

Proposed method. We propose *Double-Blind Collaborative Learning (DBCL)* as a practical defense against gradient-based attacks, e.g., (Melis et al., 2019; Zhu et al., 2019). Technically speaking, DBCL applies random sketching (Woodruff, 2014) to the parameter matrices of a neural network, and the random sketching matrices are regenerated after each iteration. Throughout the training, the clients do not see the real model parameters; what the clients see are sketched parameters. The server does not see any real gradient or descending direction; what the server sees are approximate gradients based on sketched data and sketched parameters. This is why we call our method *double-blind*.

From an honest client's perspective, DBCL is similar to dropout training, except that we use sketching to replace uniform sampling; see the discussions in Section 5.3. It is very well known that dropout does not hurt test accuracy at all (Srivastava et al., 2014; Wager et al., 2013). Since sketching has similar properties as uniform sampling, DBCL does not hurt test accuracy, which is corroborated by our empirical studies.

From an attacker's perspective, DBCL is effectively random noise injected into the gradient, which the attacker needs for inferring other clients' privacy. Roughly speaking, if a client tries to estimate the gradient, then he will get

$$\text{Estimated Grad} \;=\; \text{Transform}\big(\text{True Grad}\big) \;+\; \text{Noise}.$$

Therefore, client-side gradient-based attacks do not work. Detailed discussions are in Section 5.2.

In addition to its better security, DBCL has the following nice features:

- DBCL does not hinder test accuracy. This makes DBCL superior to the existing defenses.
- DBCL does not increase the per-iteration time complexity and communication complexity, although it reasonably increases the iterations for attaining convergence.
- When applied to dense layers and convolutional layers, DBCL does not need extra tuning.

It is worth mentioning that DBCL is not an alternative to the existing defenses; instead, DBCL can be combined with the other defenses (except dropout).

Difference from prior work. Prior work has applied sketching to neural networks for privacy, computation, or communication benefits (Hanzely et al., 2018; Li et al., 2019a). Blocki et al. (2012); Kenthapadi et al. (2012) showed that random sketching are differential private, but the papers have very different methods and applications from our work. Hanzely et al. (2018); Li et al. (2019a) applies sketching to the gradients during federated learning. In particular, Li et al. (2019a) showed that sketching the gradients is differential private.

Our method is different from the aforementioned ones. We sketch the model parameters, not the gradients. We regenerate sketching matrices after each communication. Note that sketching the gradients protects privacy at the cost of substantial worse test accuracy. In contrast, our method does not hurt test accuracy at all.

Limitations. While we propose DBCL as a practical defense against gradient-based attacks at little cost, we do not claim DBCL as a panacea. DBCL has two limitations. First, with DBCL applied, a malicious client cannot perform gradient-based attacks, but he may be able to perform parameter-based attacks such as (Hitaj et al., 2017); fortunately, the latter is much less effective than the former. Second, DBCL cannot prevent a malicious server from inferring clients' privacy, although DBCL makes the server's attack much less effective.

We admit that DBCL alone does not defeat all the attacks. To the best of our knowledge, there does not exist any defense that is effective for all the attacks that infer privacy. DBCL can be easily incorporated with existing methods such as homomorphic encryption and secret sharing to defend server-side attacks.

Paper organization. Section 2 introduces neural network, backpropagation, and matrix sketching. Section 3 defines threat models. Section 4 describes the algorithm, including the computation and communication. Section 5 theoretically studies DBCL. Section 6 presents empirical results to demonstrate that DBCL does not harm test accuracy, does not much increase the communication cost, and can defend gradient-based attacks. Section 7 discusses related work. Algorithm derivations and theoretical proofs are in the appendix. The source code is available at the Github repo: https://github.com/MengjiaoZhang/DBCL

2. Preliminaries

Dense layer. Let d_{in} be the input shape, d_{out} be the output shape, and b be the batch size. Let $\mathbf{X} \in \mathbb{R}^{b\times d_{\text{in}}}$ be the input, $\mathbf{W} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}$ be the parameter matrix, and $\mathbf{Z} = \mathbf{X}\mathbf{W}^T$ be the output. After the dense (aka fully-connected) layer, there is typically an activation function $\sigma(\mathbf{Z})$ applied elementwisely.

Backpropagation. Let L be the loss evaluated on a batch of b training samples. We derive backpropagation for the dense layer by following the convention of PyTorch. Let $\mathbf{G} \triangleq \frac{\partial L}{\partial \mathbf{Z}} \in \mathbb{R}^{b\times d_{\text{out}}}$ be the gradient received from the upper layer. We need to compute the gradients:

$$\frac{\partial L}{\partial \mathbf{X}} = \mathbf{G}\mathbf{W} \in \mathbb{R}^{b\times d_{\text{in}}} \quad \text{and} \quad \frac{\partial L}{\partial \mathbf{W}} = \mathbf{G}^T\mathbf{X} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}},$$

which can be established by the chain rule. We use $\frac{\partial L}{\partial \mathbf{W}}$ to update the parameter matrix $\mathbf{W}$ by e.g., $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$, and pass $\frac{\partial L}{\partial \mathbf{X}}$ to the lower layer.

Uniform sampling matrix. We call $\mathbf{S} \in \mathbb{R}^{d_{\text{in}} \times s}$ a uniform sampling matrix if its columns are sampled from the set $\left\{\frac{\sqrt{d_{\text{in}}}}{\sqrt{s}}\mathbf{e}_1, \cdots, \frac{\sqrt{d_{\text{in}}}}{\sqrt{s}}\mathbf{e}_{d_{\text{in}}}\right\}$ uniformly at random. Here, $\mathbf{e}_i$ is the i-th standard basis of $\mathbb{R}^{d_{\text{in}}}$. We call $\mathbf{S}$ a uniform sampling matrix because $\mathbf{XS}$ contains s randomly sampled (and scaled) columns of $\mathbf{X}$. Random matrix theories (Drineas & Mahoney, 2016; Mahoney, 2011; Martinsson & Tropp, 2020; Woodruff, 2014) guarantee that $\mathbb{E}_{\mathbf{S}}\left[\mathbf{XSS}^T\mathbf{W}^T\right] = \mathbf{XW}^T$ and that $\|\mathbf{XSS}^T\mathbf{W}^T - \mathbf{XW}^T\|$ is bounded, for any $\mathbf{X}$ and $\mathbf{W}$.

CountSketch. We call $\mathbf{S} \in \mathbb{R}^{d_{\text{in}} \times s}$ a CountSketch matrix (Charikar et al., 2004; Clarkson & Woodruff, 2013; Pham & Pagh, 2013; Thorup & Zhang, 2012; Weinberger et al., 2009) if it is constructed in the following way. Every row of $\mathbf{S}$ has exactly one nonzero entry whose position is randomly sampled from $[s] \triangleq \{1, 2, \cdots, s\}$ and value is sampled from $\{-1, +1\}$. Here is an example of $\mathbf{S}$ (10×3):

$$\mathbf{S}^T = \begin{bmatrix} 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

CountSketch has very similar properties as random Gaussian matrices (Johnson & Lindenstrauss, 1984; Woodruff, 2014). We use CountSketch for its computation efficiency. Given $\mathbf{X} \in \mathbb{R}^{b \times d_{\text{in}}}$, the CountSketch $\widetilde{\mathbf{X}} = \mathbf{XS}$ can be computed in $\mathcal{O}(d_{\text{in}}b)$ time. CountSketch is much faster than the standard matrix multiplication which has $\mathcal{O}(d_{\text{in}}bs)$ time complexity. Theories in (Clarkson & Woodruff, 2013; Meng & Mahoney, 2013; Nelson & Nguyên, 2013; Woodruff, 2014) guarantee that $\mathbb{E}_{\mathbf{S}}\left[\mathbf{XSS}^T\mathbf{W}^T\right] = \mathbf{XW}^T$ and that $\|\mathbf{XSS}^T\mathbf{W}^T - \mathbf{XW}^T\|$ is bounded, for any $\mathbf{X}$ and $\mathbf{W}$. In practice, $\mathbf{S}$ is not explicitly constructed.

3. Threat Models

In this paper, we consider attacks and defenses under the setting of client-server architecture; we assume the attacker controls a client.[1] Let $\mathbf{W}_{\text{old}}$ and $\mathbf{W}_{\text{new}}$ be the model parameters (aka weights) in two consecutive iterations. The server broadcasts $\mathbf{W}_{\text{old}}$ to the clients, the m clients use $\mathbf{W}_{\text{old}}$ and their local data to compute ascending directions $\mathbf{\Delta}_1, \cdots, \mathbf{\Delta}_m$ (e.g., gradients), and the server aggregates the directions by $\mathbf{\Delta} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{\Delta}_i$ and performs the update $\mathbf{W}_{\text{new}} \leftarrow \mathbf{W}_{\text{old}} - \mathbf{\Delta}$. Since a client (say the k-th) knows $\mathbf{W}_{\text{old}}$, $\mathbf{W}_{\text{new}}$, and his own direction $\mathbf{\Delta}_k$, he can calculate the sum of other clients' directions by

$$\begin{aligned} \sum_{i \neq k} \mathbf{\Delta}_i &= m\mathbf{\Delta} - \mathbf{\Delta}_k \\ &= m(\mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}) - \mathbf{\Delta}_k. \end{aligned} \quad (1)$$

In the case of two-party collaborative learning, that is, $m = 2$, one client knows the updating direction of the other client.

Knowing the model parameters, gradients, or both, the attacker can use various ways (Hitaj et al., 2017; Melis et al., 2019; Zhu et al., 2019) to infer other clients' privacy. We focus on gradient-based attacks (Melis et al., 2019; Zhu et al., 2019), that is, the victim's privacy is extracted from the gradients. Melis *et al.* (2019) (Melis et al., 2019) built a classifier and locally trained it for property inference. The classifier takes the updating direction $\mathbf{\Delta}_i$ as input feature and predicts the clients' data properties. The client's data cannot be recovered, however, the classifier can tell, e.g., the photo is likely female. Zhu *et al.* (2019) (Zhu et al., 2019) developed an optimization method called *gradient matching* for recovering other clients' data; both gradient and model parameters are used. It has been shown that simple defenses such as differential privacy (Dwork & Naor, 2010; Dwork, 2011) and dropout (Srivastava & Nedic, 2011) cannot defend the attacks.

In decentralized learning, where participants are compute nodes in a peer-to-peer network, a node knows its neighbors' model parameters and thereby updating directions. A node can infer the privacy of its neighbors in the same way as (Melis et al., 2019; Zhu et al., 2019). In Appendix D, we discuss the attack and defense in decentralized learning; they will be our future work.

4. Proposed Method

We present the high-level ideas in Section 4.1, elaborate on the implementation in Section 4.2, and analyze the time and communication complexities in Section 4.3.

4.1. High-Level Ideas

The attacks of (Melis et al., 2019; Zhu et al., 2019) need the victim's updating direction, e.g., gradient, for inferring the victim's privacy. Using standard distributed algorithms such as distributed SGD and Federated Averaging (`FedAvg`) (McMahan et al., 2017), the server can see the clients' updating directions, $\mathbf{\Delta}_1, \cdots, \mathbf{\Delta}_m$, and the clients can see the jointly learned model parameter, $\mathbf{W}$. A malicious client can use (1) to get other clients' updating directions and then perform the gradient-based attacks such as (Melis et al., 2019; Zhu et al., 2019).

To defend the gradient-based attacks, our proposed Double-Blind Collaborative Learning (DBCL) applies random

[1] A stronger assumption would be that the server is malicious. Our defense may not defeat a malicious server.

sketching to the inputs and parameter matrices. Let $\mathbf{X}$ and $\mathbf{W}$ be the input batch and parameter matrix of a dense layer, respectively. For some or all the dense layers, replace $\mathbf{X}$ and $\mathbf{W}$ by $\widetilde{\mathbf{X}} = \mathbf{XS}$ and $\widetilde{\mathbf{W}} = \mathbf{WS}$, respectively, and different layers have different sketching matrices, $\mathbf{S}$. Each time $\mathbf{W}$ is updated, we re-generate $\mathbf{S}$. DBCL is applicable to convolutional layers in a similar way; see Appendix B.2.

Let $\mathbf{W}_{\text{old}}$ and $\mathbf{W}_{\text{new}}$ be the true parameter matrices in two consecutive iterations; they are known to only the server. The clients do not observe $\mathbf{W}_{\text{old}}$ and $\mathbf{W}_{\text{new}}$. What the clients observe are the random sketches: $\widetilde{\mathbf{W}}_{\text{old}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}$ and $\widetilde{\mathbf{W}}_{\text{new}} = \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}$. In addition, the clients know $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$. To perform gradient-based attacks, a client seeks to estimate $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$ based on what it knows, i.e., $\widetilde{\mathbf{W}}_{\text{old}}$, $\widetilde{\mathbf{W}}_{\text{new}}$, $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$. Our theories and experiments show that a client's estimate of $\mathbf{\Delta}$ is far from the truth.

4.2. Algorithm Description

We describe the computation and communication operations of DBCL. We consider the client-server architecture, dense layers, and the distributed SGD algorithm.[2] DBCL works in the following four steps. Broadcasting and aggregation are communication operations; forward pass and backward pass are local computations performed by each client for calculating gradients.

Broadcasting. The central server generates a new seed ψ[3] and then a random sketch: $\widetilde{\mathbf{W}} = \mathbf{WS}$. It broadcasts ψ and $\widetilde{\mathbf{W}} \in \mathbb{R}^{d_{\text{out}} \times s}$ to all the clients through message passing. Here, the sketch size s is determined by the server and must be set smaller than d_{in}; the server may vary s after each iteration.

Local forward pass. The i-th client randomly selects a batch of b samples from its local dataset and then locally performs a forward pass. Let the input of a dense layer be $\mathbf{X}_i \in \mathbb{R}^{b \times d_{\text{in}}}$. The client uses the seed ψ to draw a sketch $\widetilde{\mathbf{X}}_i = \mathbf{X}_i\mathbf{S} \in \mathbb{R}^{b \times s}$ and computes $\mathbf{Z}_i = \widetilde{\mathbf{X}}_i\widetilde{\mathbf{W}}^T$. Then $\sigma(\mathbf{Z}_i)$ becomes the input of the upper layer, where σ is some activation function. Repeat this process for all the layers. The forward pass finally outputs L_i, the loss evaluated on the batch of b samples.

Local backward pass. Let the local gradient propagated to the dense layer be $\mathbf{G}_i \triangleq \frac{\partial L_i}{\partial \mathbf{Z}_i} \in \mathbb{R}^{b \times d_{\text{out}}}$. The client locally calculates

$$\mathbf{\Gamma}_i = \mathbf{G}_i^T\widetilde{\mathbf{X}}_i \in \mathbb{R}^{d_{\text{out}} \times s} \text{ and } \frac{\partial L_i}{\partial \mathbf{X}_i} = \mathbf{G}_i\widetilde{\mathbf{W}}\mathbf{S}^T \in \mathbb{R}^{b \times d_{\text{in}}}.$$

The gradient $\frac{\partial L_i}{\partial \mathbf{X}_i}$ is propagated to the lower-level layer to continue the backpropagation.

Aggregation. The server aggregates $\{\mathbf{\Gamma}_i\}_{i=1}^m$ to compute $\mathbf{\Gamma} = \frac{1}{m}\sum_{i=1}^m \mathbf{\Gamma}_i$; this needs a communication. Let $L = \frac{1}{m}\sum_{i=1}^m L_i$ be the loss evaluated on the batch of mb samples. It can be shown that

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{m}\sum_{i=1}^m \frac{\partial L_i}{\partial \mathbf{W}} = \mathbf{\Gamma}\mathbf{S}^T \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}. \quad (2)$$

The server then updates the parameters by, e.g., $\mathbf{W} \leftarrow \mathbf{W} - \eta\frac{\partial L}{\partial \mathbf{W}}$.

4.3. Time Complexity and Communication Complexity

DBCL does not increase the time complexity of local computations. The CountSketch, $\widetilde{\mathbf{X}}_i = \mathbf{X}_i\mathbf{S}$ and $\widetilde{\mathbf{W}}_i = \mathbf{W}_i\mathbf{S}$, costs $\mathcal{O}(bd_{\text{in}})$ and $\mathcal{O}(d_{\text{in}}d_{\text{out}})$ time, respectively. Using CountSketch, the overall time complexity of a forward and a backward pass is $\mathcal{O}(bd_{\text{in}} + d_{\text{in}}d_{\text{out}} + bsd_{\text{out}})$. Since we set $s < d_{\text{in}}$ to protect privacy, the time complexity is lower than the standard backpropagation, $\mathcal{O}(bd_{\text{in}}d_{\text{out}})$.

DBCL does not increase per-iteration communication complexity. Without using sketching, the communicated matrices are $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and $\frac{\partial L_i}{\partial \mathbf{W}} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$. Using sketching, the communicated matrices are $\widetilde{\mathbf{W}} \in \mathbb{R}^{d_{\text{out}} \times s}$ and $\mathbf{\Gamma}_i \in \mathbb{R}^{d_{\text{out}} \times s}$. Because $s < d_{\text{in}}$, the per-iteration communication complexity is lower than the standard distributed SGD.

5. Theoretical Insights

In Section 5.1, we discuss how a malicious client makes use of the sketched model parameters for privacy inference. In Sections 5.2, we show that DBCL can defend certain types of attacks. In Section 5.3, we give an explanation of DBCL from optimization perspective.

5.1. Approximating the Gradient

Assume the attacker controls a client and participates in collaborative learning. Let $\mathbf{W}_{\text{old}}$ and $\mathbf{W}_{\text{new}}$ be the parameter matrices of two consecutive iterations; they are unknown to the clients. What a client sees are the sketches, $\widetilde{\mathbf{W}}_{\text{old}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}$ and $\widetilde{\mathbf{W}}_{\text{new}} = \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}$. To conduct gradient-based attacks, the attacker must know the gradient $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$.

Without using $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$, one cannot recover $\mathbf{\Delta}$, not even approximately. The difference between sketched parameters, $\widetilde{\mathbf{\Delta}} = \widetilde{\mathbf{W}}_{\text{old}} - \widetilde{\mathbf{W}}_{\text{new}}$, is entirely different from

[2] DBCL works also for convolutional layers; see the appendix for the details. DBCL can be easily extended to `FedAvg` or other communication-efficient frameworks. DBCL can be applied to peer-to-peer networks; see the discussions in the appendix.

[3] Let the clients use the same pseudo-random number generator as the server. Given the seed ψ, all the clients can construct the same sketching matrix $\mathbf{S}$.

the real gradient, $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$.[4] We can even vary the sketch size s with iterations so that $\widetilde{\mathbf{W}}_{\text{old}}$ and $\widetilde{\mathbf{W}}_{\text{new}}$ have different number of columns, making it impossible to compute $\mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$.

Note that the clients know also $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$. A smart attacker, who knows random matrix theory, may want to use

$$\widehat{\mathbf{\Delta}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T$$

to approximate $\mathbf{\Delta}$. It is because $\widehat{\mathbf{\Delta}}$ is an unbiased estimate of $\mathbf{\Delta}$, i.e., $\mathbb{E}[\widehat{\mathbf{\Delta}}] = \mathbf{\Delta}$, where the expectation is taken w.r.t. the random sketching matrices $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$.

5.2. Defending Gradient-Based Attacks

We analyze the attack that uses $\widehat{\mathbf{\Delta}}$. We first give an intuitive explanation and then prove that using $\widehat{\mathbf{\Delta}}$ does not work, unless the magnitude of $\mathbf{\Delta}$ is smaller than $\mathbf{W}_{\text{new}}$.

Matrix sketching as implicit noise. As $\widehat{\mathbf{\Delta}}$ is an unbiased estimate of $\mathbf{\Delta}$, the reader may wonder why $\widehat{\mathbf{\Delta}}$ does not disclose the information of $\mathbf{\Delta}$. Here we give an intuitive explanation. $\widehat{\mathbf{\Delta}}$ is a mix of $\mathbf{\Delta}$ (which is the signal) and a random transformation of $\mathbf{W}_{\text{new}}$ (which is random noise):

$$\widehat{\mathbf{\Delta}} = \underbrace{\mathbf{\Delta}}_{\text{signal}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T + \underbrace{\mathbf{W}_{\text{new}}\left(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\right)}_{\text{zero-mean noise}}. \quad (3)$$

As the magnitude of $\mathbf{W}$ is much greater than $\mathbf{\Delta}$,[5] the noise outweighs the signal, making $\widehat{\mathbf{\Delta}}$ far from $\mathbf{\Delta}$. From the attacker's perspective, random sketching is effectively random noise that outweighs the signal.

Defending the gradient matching attack of (Zhu et al., 2019). The gradient matching attack can recover the victim's original data based on the victim's gradient, $\mathbf{\Delta}_i$, and the model parameters, $\mathbf{W}$. Numerical optimization is used to find the data on which the computed gradient matches $\mathbf{\Delta}_i$. To get $\mathbf{\Delta}_i$, the attacker must know $\mathbf{\Delta} = \sum_{i=1}^m \mathbf{\Delta}_i$. Using DBCL, no client knows $\mathbf{\Delta}$. A smart attacker may want to use $\widehat{\mathbf{\Delta}}$ in lieu of $\mathbf{\Delta}$ because of its unbiasedness. We show in Theorem 1 that this approach does not work.

Theorem 1. *Let $\mathbf{S}_{new}$ and $\mathbf{S}_{new}$ be $d_{in} \times s$ CountSketch matrices and $s < d_{in}$. Then*

$$\mathbb{E}\left\|\widehat{\mathbf{\Delta}} - \mathbf{\Delta}\right\|_F^2 = \Omega\left(\frac{d_{in}}{s}\right) \cdot \left(\left\|\mathbf{W}_{old}\right\|_F^2 + \left\|\mathbf{W}_{new}\right\|_F^2\right).$$

[4] The columns of $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$ are randomly permuted. Even if $\widetilde{\mathbf{\Delta}}$ is close to $\mathbf{\Delta}$, after randomly permuting the columns of $\mathbf{S}_{\text{old}}$ or $\mathbf{S}_{\text{new}}$, $\widetilde{\mathbf{\Delta}}$ becomes entirely different.

[5] In machine learning, $\mathbf{\Delta}$ is the updating direction, e.g., gradient. The magnitude of gradient is much smaller than the model parameters $\mathbf{W}$, especially when $\mathbf{W}$ is close to a stationary point.

Since the magnitude of $\mathbf{\Delta}$ is typically smaller than $\mathbf{W}$, Theorem 1 guarantees that using $\widehat{\mathbf{\Delta}}$ is no better than all-zeros or random guessing.

Remark 1. *The theorem shows that DBCL beats one way of gradient estimate. Admittedly, the theorem does not guarantee that DBCL can defend all kinds of gradient estimates. A stronger bound would be:*

$$\min_{\mathbf{Z}_1,\mathbf{Z}_2} \left\|\mathbf{W}_{old}\mathbf{S}_{old}\mathbf{Z}_1 - \mathbf{W}_{new}\mathbf{S}_{new}\mathbf{Z}_2 - \mathbf{\Delta}\right\|_F^2 = \Omega(1)\cdot\left\|\mathbf{\Delta}\right\|_F^2.$$

Unfortunately, at this time, we do not know how to prove such a bound.

Defending the property inference attack (PIA) of (Melis et al., 2019). To conduct the PIA of (Melis et al., 2019), the attacker may want to use a linear model parameterized by $\mathbf{V}$.[6] According to (1), the attacker uses $\mathbf{\Delta} - \mathbf{A}$ as input features for PIA, where $\mathbf{A}$ is some fixed matrix known to the attacker. The linear model makes prediction by $\mathbf{Y} \triangleq (\mathbf{\Delta} - \mathbf{A})\mathbf{V}^T$. Using $\widehat{\mathbf{\Delta}}$ to approximate $\mathbf{\Delta}$, the prediction is $\widehat{\mathbf{Y}} \triangleq (\widehat{\mathbf{\Delta}} - \mathbf{A})\mathbf{V}^T$. Theorem 2 and Corollary 3 show that $\|\widehat{\mathbf{Y}} - \mathbf{Y}\|_F^2 = \|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\|_F^2$ is very big.

Theorem 2. *Let $\mathbf{S}_{old}$ and $\mathbf{S}_{new}$ be $d_{in} \times s$ CountSketch matrices and $s < d_{in}$. Let w_{pq} be the (p,q)-th entry of $\mathbf{W}_{old} \in \mathbb{R}^{d_{out} \times d_{in}}$ and $\tilde{w}_{pq}$ be the (p,q)-th entry of $\mathbf{W}_{new} \in \mathbb{R}^{d_{out} \times d_{in}}$. Let $\mathbf{V}$ be any $r \times d_{in}$ matrix and v_{pq} be the (p,q)-th entry of $\mathbf{V}$. Then*

$$\mathbb{E}\left\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\right\|_F^2 = \frac{1}{s}\sum_{i=1}^{d_{out}}\sum_{j=1}^{r}\sum_{k\neq l}\left(w_{ik}^2 v_{jl}^2 + w_{ik}v_{jk}w_{il}v_{jl} + \tilde{w}_{ik}^2 v_{jl}^2 + \tilde{w}_{ik}v_{jk}\tilde{w}_{il}v_{jl}\right).$$

The bound in Theorem 2 is involved. To interpret the bound, we add (somehow unrealistic) assumptions and obtain Corollary 3.

Corollary 3. *Let $\mathbf{S}$ be a $d_{in} \times s$ CountSketch matrix and $s < d_{in}$. Assume that the entries of $\mathbf{W}_{old}$ are IID and that the entries of $\mathbf{V}$ are also IID. Then*

$$\mathbb{E}\left\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\right\|_F^2 = \Omega\left(\frac{d_{in}}{s}\right) \cdot \left\|\mathbf{W}_{old}\mathbf{V}^T\right\|_F^2.$$

Since the magnitude of $\mathbf{\Delta}$ is much smaller than $\mathbf{W}$, especially when $\mathbf{W}$ is close to a stationary point, $\|\mathbf{W}\mathbf{V}^T\|_F^2$ is typically greater than $\|\mathbf{\Delta}\mathbf{V}^T\|_F^2$. Thus, $\mathbb{E}\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\|_F^2$ is typically bigger than $\|\mathbf{\Delta}\mathbf{V}^T\|_F^2$, which implies that using $\widehat{\mathbf{\Delta}}$ is no better than all-zeros or random guessing.

[6] The conclusion applies also to neural networks because its first layer is such a linear model.

Table 1. Experiments on MNIST. The table shows the rounds of communications for attaining the test accuracy. Here, c is the participation ratio of `FedAvg`, that is, in each round, only a fraction of clients participate in the training.

Models	Accuracy	Communication Rounds				
		$c = 1\%$	$c = 10\%$	$c = 20\%$	$c = 50\%$	$c = 100\%$
MLP	0.97	222	96	84	83	82
MLP-Sketch	0.97	572	322	308	298	287
CNN	0.99	462	309	97	91	31
CNN-Sketch	0.99	636	176	189	170	174

5.3. Understanding DBCL from Optimization Perspective

We give an explanation of DBCL from optimization perspective. Let us consider the generalized linear model:

$$\operatorname*{argmin}_{\mathbf{w}} \left\{ f(\mathbf{w}) \triangleq \frac{1}{n} \sum_{j=1}^{n} \ell\big(\mathbf{x}_i^T \mathbf{w},\, y_j\big) \right\}, \tag{4}$$

where $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)$ are the training samples and $\ell(\cdot, \cdot)$ is the loss function. If we apply sketching to a generalized linear model, then the training will be solving the following problem:

$$\operatorname*{argmin}_{\mathbf{w}} \left\{ \tilde{f}(\mathbf{w}) \triangleq \mathbb{E}_{\mathbf{S}} \left[\frac{1}{n} \sum_{j=1}^{n} \ell\big(\mathbf{x}_i^T \mathbf{S}\mathbf{S}^T \mathbf{w},\, y_j\big) \right] \right\}. \tag{5}$$

Note that (5) is different from (4). If $\mathbf{S}$ is a uniform sampling matrix, then (5) will be empirical risk minimization with dropout. Prior work (Wager et al., 2013) proved that dropout is equivalent to adaptive regularization which can alleviate overfitting. Random projections such as CountSketch have the same properties as uniform sampling (Woodruff, 2014), and thus the role of random sketching in (5) can be thought of as adaptive regularization. This is why DBCL does not hinder prediction accuracy at all.

6. Experiments

We conduct experiments to demonstrate that first, DBCL does not harm test accuracy, second, DBCL does not increase the communication cost too much, and third, DBCL can defend client-side gradient-based attacks.

6.1. Experiment Setting

Our method and the compared methods are implemented using PyTorch. The experiments are conducted on a server with 4 NVIDIA GeForce Titan V GPUs, 2 Xeon Gold 6134 CPUs, and 192 GB memory. We follow the settings of the relevant papers to perform comparisons.

Three datasets are used in the experiments. MNIST has 60,000 training images and 10,000 test images; each image is 28×28. CIFAR-10 has 50,000 training images and 10,000 test images; each image is $32 \times 32 \times 3$. Labeled Faces In the Wild (LFW) has 13,233 faces of 5,749 individuals; each face is a $64 \times 47 \times 3$ color image.

6.2. Accuracy and Efficiency

We conduct experiments on the MNIST and CIFAR-10 datasets to show that first, DBCL does not hinder prediction accuracy, and second, it does not much increase the communication cost. We follow the setting of (McMahan et al., 2017). The learning rates are tuned to optimize the convergence rate.

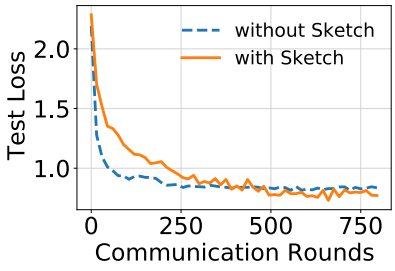
MNIST classification. We build a multilayer perceptron (MLP) and a convolutional neural network (CNN) for the multi-class classification task. The MLP has 3 dense layers: $\mathsf{Dense}(200) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(200) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(10) \Rightarrow \mathsf{Softmax}$. The CNN has 2 convolutional layers and 2 dense layers: $\mathsf{Conv}(32, 5 \times 5) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2 \times 2) \Rightarrow \mathsf{Conv}(64, 5 \times 5) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2 \times 2) \Rightarrow \mathsf{Flatten} \Rightarrow \mathsf{Dense}(512) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(10) \Rightarrow \mathsf{Softmax}$.

We use Federated Averaging (`FedAvg`) to train the MLP and CNN. The data are partitioned among 100 (virtual) clients uniformly at random. Between two communications, `FedAvg` performs local computation for 1 epoch (for MLP) or 5 epochs (for CNN). The batch size of local SGD is set to 10.

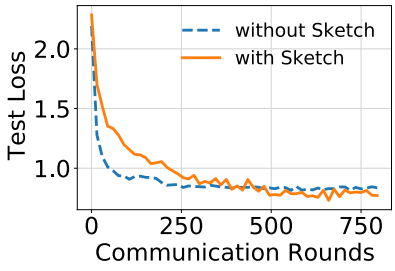
Sketching is applied to all the dense and convolutional layers except the output layer. We set the sketch size to $s = d_{\text{in}}/2$; thus, the per-communication word complexity is reduced by half. With sketching, the MLP and CNN are trained by `FedAvg` under the same setting.

We show the experimental results in Table 1. Trained by `FedAvg`, the small MLP can only reach 97% test accuracy, while the CNN can obtain 99% test accuracy. In this set of experiments, sketching does not hinder test accuracy at all. We show the rounds of communications for attaining the test accuracies. For the MLP, sketching needs 2.6x $\sim$ 3.6x rounds of communications to converge. For the CNN, sketching needs 0.6x $\sim$ 5.6x rounds of communications to converge. Using sketching, the per-communication word complexity is reduced to 0.5x.

communication rounds ↑
per-communication word complexity ↓.



(a) Test loss.



(b) Test Accuracy.

Figure 1. Experiment on CIFAR-10 dataset. The test accuracy do not match the state-of-the-art because the CNN is small and we do not use advanced tricks; we follow the settings of the seminal work (McMahan et al., 2017).

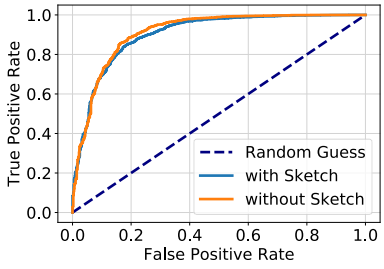


Figure 2. Gender classification on the LWF dataset.

CIFAR-10 classification. We build a CNN with 3 convolutional layers and 2 dense layers: $\mathsf{Conv}(32, 5\times 5) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Conv}(64, 5\times 5) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2\times 2) \Rightarrow \mathsf{Conv}(128, 5\times 5) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2\times 2) \Rightarrow \mathsf{Flatten} \Rightarrow \mathsf{Dense}(200) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(10) \Rightarrow \mathsf{Softmax}$.

The CNN is also trained using `FedAvg`. The data are partitioned among 100 clients. We set the participation ratio to $c = 10\%$, that is, each time only 10% uniformly sampled clients participate in the training. Between two communications, `FedAvg` performs local computation for 5 epochs. The batch size of local SGD is set to 50. We follow the settings of (McMahan et al., 2017), so we do not use tricks such as data augmentation, batch normalization, skip connection, etc.

Figure 1 shows the convergence curves. Using sketching does not hinder the test accuracy at all; on the contrary, it marginally improves the test accuracy. The reason is likely that sketching is an adaptive regularization similar to dropout; see the discussions in Section 5.3.

Binary classification on imbalanced data. Following (Melis et al., 2019), we conduct binary classification experiments on a subset of the LFW dataset. We use $8,150$ faces for training and $3,400$ for test. The task is gender prediction. We build a CNN with 3 convolutional layers and 3 dense layers: $\mathsf{Conv}(64, 3\times 3) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2\times 2) \Rightarrow \mathsf{Conv}(64, 3\times 3) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2\times 2) \Rightarrow \mathsf{Conv}(128, 3\times 3) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{MaxPool}(2\times 2) \Rightarrow \mathsf{Flatten} \Rightarrow \mathsf{Dense}(32) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(32) \Rightarrow \mathsf{ReLU} \Rightarrow \mathsf{Dense}(1) \Rightarrow \mathsf{Sigmoid}$. We apply sketching to all the convolutional and dense layers except the output layer. The model is trained by distributed SGD (2 clients and 1 server) with a learning rate of 0.01 and a batch size of 32.

The dataset is class-imbalanced: 8957 are males, and 2593 are females. We therefore use ROC curves, instead of classification accuracy, for evaluating the classification. In Figure 2, we plot the ROC curves to compare the standard CNN and the sketched one. The two ROC curves are almost the same.

6.3. Defending Gradient-Based Attacks

We empirically study whether DBCL can defend the gradient-based attacks of (Melis et al., 2019) and (Zhu et al., 2019). The details of experiment setting are in the appendix.

Gradient estimation. To perform gradient-based attacks, a client needs to (approximately) know the gradient (or other updating directions). We discuss in Section 5.1 how a client can estimate the updating directions. We empirically study two approaches:

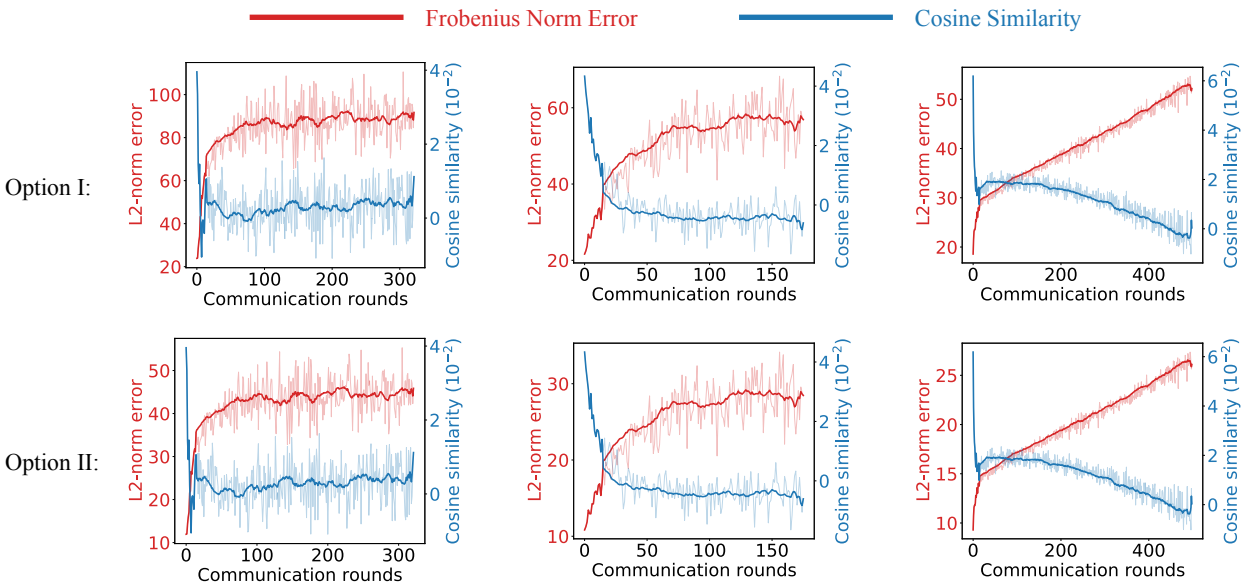
$$\text{Option I:}\quad \widehat{\mathbf{\Delta}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T,$$
$$\text{Option II:}\quad \widehat{\mathbf{\Delta}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^\dagger - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^\dagger.$$

Here, $\mathbf{A}^\dagger$ means the Moore-Penrose inverse of matrix $\mathbf{A}$. Let $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$ be the true updating direction. We evaluate the approximation quality using two metrics:

$$\text{The } \ell_2\text{-norm error:}\quad \left\|\mathrm{vec}\left(\widehat{\mathbf{\Delta}} - \mathbf{\Delta}\right)\right\|_2 \Big/ \left\|\mathrm{vec}\left(\mathbf{\Delta}\right)\right\|_2,$$
$$\text{Cosine similarity:}\quad \left\langle\, \mathrm{vec}\left(\widehat{\mathbf{\Delta}}\right),\, \mathrm{vec}\left(\mathbf{\Delta}\right)\,\right\rangle.$$

If $\widehat{\mathbf{\Delta}}$ is far from $\mathbf{\Delta}$, which means our defense is effective, then the ℓ_2 error is big, and the cosine similarity is small.

Figure 3 plots the quality of gradient estimation. The experiment settings are the same as Section 6.2; the participation ratio is set to 10%. The results show that $\widehat{\mathbf{\Delta}}$ is entirely different from $\mathbf{\Delta}$, which means our defense works. Equation (3) implies that as the magnitude of $\mathbf{\Delta}$ decreases, $\widehat{\mathbf{\Delta}}$ is dominated by noise, and thus the error $\frac{\|\mathrm{vec}(\widehat{\mathbf{\Delta}}-\mathbf{\Delta})\|_2}{\|\mathrm{vec}(\mathbf{\Delta})\|_2}$ gets bigger. In our experiments, as the communication rounds increase, the algorithm tends to converge, the magnitude of $\mathbf{\Delta}$ decreases, and the error increases. The empirical observations verify our theories.



(a) MLP trained on MNIST. (b) CNN trained on MNIST. (c) CNN trained on CIFAR-10.

Figure 3. The x-axis is the communication rounds. The y-axes are Frobenius norm errors (red) and cosine similarities (blue). The figures show that the estimated gradient, $\widehat{\mathbf{\Delta}}$, is far from the true gradient, $\mathbf{\Delta}$, which means our defense works.

Defending the property inference attack (PIA) of Melis et al. (2019). We conduct experiments on the LFW dataset. The task of collaborative learning is the gender classification under the same setting as Section 6.2. The attacker seeks to infer whether a single batch of photos in the victim's private dataset contain Africans or not. We use one server and two clients, which is the easiest for the attacker.

- Let one client be the attacker and the other client be the victim. Without sketching, the AUC is 1.0, which means the attacker is always right. With sketching applied, the AUC is 0.5, which means the performance is the same as random guess.
- Let the server be the attacker and one client be the victim. Without sketching, the AUC is 1.0, which means the attacker succeeds. With sketching applied, the AUC is 0.726. Our defense makes server-side attack much less effective, but users' privacy can still leak.

Defending the gradient-matching attack of Zhu et al. (2019). The attack seeks to recover the victims' data using model parameters and gradients. They seek to find a batch of images by optimization so that the resulting gradient matches the observed gradient of the victim. We use the same CNNs as (Zhu et al., 2019) to conduct experiments on the MNIST and CIFAR-10 datasets. Without using sketching, the gradient-matching attack very well recovers the images. For both client-side attacks and server-side attacks, if sketching is applied to all except the output layer, then the recovered images are just like random noise. The experiments show that we can defend gradient-matching attack performed by both server and client.

7. Related Work

Cryptography approaches such as secure aggregation (Bonawitz et al., 2017), homomorphic encryption (Aono et al., 2017; Giacomelli et al., 2018; Gilad-Bachrach et al., 2016; Liu et al., 2018; Zhang et al., 2018b), Yao's garbled circuit protocol (Rouhani et al., 2018), and many other methods (Yuan & Yu, 2013; Zhang et al., 2018a) can also improve the security of collaborative learning. Generative models such as (Chen et al., 2018) can also improve privacy; however, they hinder the accuracy and efficiency, and their tuning and deployment are nontrivial. All the mentioned defenses are not competitive methods of our DBCL; instead, they can be combined with DBCL to defend more attacks.

Our methodology is based on matrix sketching (Johnson & Lindenstrauss, 1984; Drineas et al., 2008; Halko et al., 2011; Mahoney, 2011; Woodruff, 2014; Drineas & Mahoney, 2016). Sketching has been applied to achieve differential privacy (Blocki et al., 2012; Kenthapadi et al., 2012; Li et al., 2019a). It has been shown that to protect privacy, matrix sketching has the same effect as injecting random noise. Our method is developed based on the connection between sketching (Woodruff, 2014) and dropout training

(Srivastava et al., 2014); in particular, if $\mathbf{S}$ is uniform sampling, then DBCL is essentially dropout. Our approach is similar to the contemporaneous work (Khaled & Richtárik, 2019) which is developed for computational benefits.

Decentralized learning, that is, the clients perform peer-to-peer communication without a central server, is an alternative to federated learning and has received much attention in recent years (Colin et al., 2016; Koloskova et al., 2019; Lan et al., 2017; Luo et al., 2019; Ram et al., 2010; Sirb & Ye, 2016; Tang et al., 2018; Wang et al., 2019; Yuan et al., 2016). The attacks of Melis et al. (2019); Zhu et al. (2019) can be applied to decentralized learning, and DBCL can defend the attacks under the decentralized setting. We discuss decentralized learning in the appendix. The attacks and defense under the decentralized setting will be our future work.

8. Conclusions

Collaborative learning enables multiple parties to jointly train a model without data sharing. Unfortunately, standard distributed optimization algorithms can easily leak participants' privacy. We proposed Double-Blind Collaborative Learning (DBCL) for defending gradient-based attacks which are the most effective privacy inference methods. We showed that DBCL can defeat gradient-based attacks conducted by malicious clients. Admittedly, DBCL can not defend all kinds of attacks; for example, if the server is malicious, then the attack of (Melis et al., 2019) still works, but much less effectively. While it improves privacy, DBCL does not hurt test accuracy at all and does not much increase the cost of training. DBCL is easy to use and does not need extra tuning. Our future work will combine DBCL with cryptographic methods such as homomorphic encryption and secret sharing so that neither client nor server can infer users' privacy.

Acknowledgements

The author thanks Michael Mahoney, Richard Peng, Peter Richtárik, and David Woodruff for their helpful suggestions.

References

Aono, Y., Hayashi, T., Wang, L., Moriai, S., et al. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2017.

Ateniese, G., Mancini, L. V., Spognardi, A., Villani, A., Vitali, D., and Felici, G. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks*, 10(3):137–150, September 2015. ISSN 1747-8405.

Bianchi, P., Fort, G., and Hachem, W. Performance of a distributed stochastic approximation algorithm. *IEEE Transactions on Information Theory*, 59(11):7405–7418, 2013.

Blocki, J., Blum, A., Datta, A., and Sheffet, O. The Johnson-Lindenstrauss transform itself preserves differential privacy. In *Annual Symposium on Foundations of Computer Science (FOCS)*, 2012.

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A., and Seth, K. Practical secure aggregation for privacy preserving machine learning. *IACR Cryptology ePrint Archive*, 2017:281, 2017.

Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

Chen, Q., Xiang, C., Xue, M., Li, B., Borisov, N., Kaarfar, D., and Zhu, H. Differentially private data generative models. *arXiv preprint arXiv:1812.02274*, 2018.

Clarkson, K. L. and Woodruff, D. P. Low rank approximation and regression in input sparsity time. In *Annual ACM Symposium on theory of computing (STOC)*, 2013.

Colin, I., Bellet, A., Salmon, J., and Clémençon, S. Gossip dual averaging for decentralized optimization of pairwise functions. *arXiv preprint arXiv:1606.02421*, 2016.

Drineas, P. and Mahoney, M. W. RandNLA: randomized numerical linear algebra. *Communications of the ACM*, 59(6):80–90, 2016.

Drineas, P., Mahoney, M. W., and Muthukrishnan, S. Relative-error CUR matrix decompositions. *SIAM Journal on Matrix Analysis and Applications*, 30(2):844–881, September 2008.

Dwork, C. Differential privacy. *Encyclopedia of Cryptography and Security*, pp. 338–340, 2011.

Dwork, C. and Naor, M. On the difficulties of disclosure prevention in statistical databases or the case for differential privacy. *Journal of Privacy and Confidentiality*, 2(1), 2010.

Fredrikson, M., Jha, S., and Ristenpart, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

Giacomelli, I., Jha, S., Joye, M., Page, C. D., and Yoon, K. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *International Conference on Applied Cryptography and Network Security*, pp. 243–261. Springer, 2018.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning (ICML)*, 2016.

Halko, N., Martinsson, P.-G., and Tropp, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

Hanzely, F., Mishchenko, K., and Richtárik, P. SEGA: Variance reduction via gradient sketching. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Hitaj, B., Ateniese, G., and Perez-Cruz, F. Deep models under the GAN: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

Johnson, W. B. and Lindenstrauss, J. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206), 1984.

Kenthapadi, K., Korolova, A., Mironov, I., and Mishra, N. Privacy via the Johnson-Lindenstrauss transform. *arXiv preprint arXiv:1204.2606*, 2012.

Khaled, A. and Richtárik, P. Gradient descent with compressed iterates. *arXiv*, 2019.

Koloskova, A., Stich, S. U., and Jaggi, M. Decentralized stochastic optimization and gossip algorithms with compressed communication. *arXiv preprint arXiv:1902.00340*, 2019.

Lan, G., Lee, S., and Zhou, Y. Communication-efficient algorithms for decentralized and stochastic optimization. *Mathematical Programming*, pp. 1–48, 2017.

Li, T., Liu, Z., Sekar, V., and Smith, V. Privacy for free: Communication-efficient learning with differential privacy using sketches. *arXiv preprint arXiv:1911.00972*, 2019a.

Li, X., Huang, K., Yang, W., Wang, S., and Zhang, Z. On the convergence of FedAvg on Non-IID data. *arXiv:1907.02189*, 2019b.

Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., and Liu, J. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

Liu, Y., Chen, T., and Yang, Q. Secure federated transfer learning. *arXiv preprint arXiv:1812.03337*, 2018.

Luo, Q., He, J., Zhuo, Y., and Qian, X. Heterogeneity-aware asynchronous decentralized training. *arXiv preprint arXiv:1909.08029*, 2019.

Mahoney, M. W. Randomized algorithms for matrices and data. *Foundations and Trends in Machine Learning*, 3(2): 123–224, 2011.

Martinsson, P.-G. and Tropp, J. Randomized numerical linear algebra: Foundations & algorithms. *arXiv preprint arXiv:2002.01387*, 2020.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics (AISTATS)*, 2017.

Melis, L., Song, C., De Cristofaro, E., and Shmatikov, V. Exploiting unintended feature leakage in collaborative learning. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

Meng, X. and Mahoney, M. W. Low-Distortion Subspace Embeddings in Input-Sparsity Time and Applications to Robust Linear Regression. In *Annual ACM Symposium on Theory of Computing (STOC)*, 2013.

Nelson, J. and Nguyên, H. L. OSNAP: Faster Numerical Linear Algebra Algorithms via Sparser Subspace Embeddings. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2013.

Pham, N. and Pagh, R. Fast and scalable polynomial kernels via explicit feature maps. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

Ram, S. S., Nedić, A., and Veeravalli, V. V. Asynchronous gossip algorithm for stochastic optimization: Constant stepsize analysis. In *Recent Advances in Optimization and its Applications in Engineering*, pp. 51–60. Springer, 2010.

Rouhani, B. D., Riazi, M. S., and Koushanfar, F. DeepSecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, 2018.

Sahu, A. K., Li, T., Sanjabi, M., Zaheer, M., Talwalkar, A., and Smith, V. Federated optimization for heterogeneous networks. *arXiv preprint arXiv:1812.06127*, 2019.

Sirb, B. and Ye, X. Consensus optimization with delayed and stochastic gradients on decentralized networks. In *2016 IEEE International Conference on Big Data (Big Data)*, pp. 76–85. IEEE, 2016.

Srivastava, K. and Nedic, A. Distributed asynchronous constrained stochastic optimization. *IEEE Journal of Selected Topics in Signal Processing*, 5(4):772–790, 2011.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Stich, S. U. Local SGD converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

Tang, H., Lian, X., Yan, M., Zhang, C., and Liu, J. D2: Decentralized training over decentralized data. *arXiv preprint arXiv:1803.07068*, 2018.

Thorup, M. and Zhang, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM Journal on Computing*, 41(2): 293–331, April 2012. ISSN 0097-5397.

Wager, S., Wang, S., and Liang, P. S. Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

Wang, J. and Joshi, G. Cooperative SGD: A unified framework for the design and analysis of communication-efficient SGD algorithms. *arXiv preprint arXiv:1808.07576*, 2018.

Wang, J., Sahu, A. K., Yang, Z., Joshi, G., and Kar, S. Matcha: Speeding up decentralized sgd via matching decomposition sampling. *arXiv preprint arXiv:1905.09435*, 2019.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. Feature hashing for large scale multitask learning. In *International Conference on Machine Learning (ICML)*, 2009.

Woodruff, D. P. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

Yu, H., Yang, S., and Zhu, S. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI Conference on Artificial Intelligence*, 2019.

Yuan, J. and Yu, S. Privacy preserving back-propagation neural network learning made practical with cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):212–221, 2013.

Yuan, K., Ling, Q., and Yin, W. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.

Zhang, D., Chen, X., Wang, D., and Shi, J. A survey on collaborative deep learning and privacy-preserving. In *IEEE Third International Conference on Data Science in Cyberspace (DSC)*, 2018a.

Zhang, Q., Wang, C., Wu, H., Xin, C., and Phuong, T. V. GELU-Net: A globally encrypted, locally unencrypted deep neural network for privacy-preserved learning. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2018b.

Zhou, F. and Cong, G. On the convergence properties of a k-step averaging stochastic gradient descent algorithm for nonconvex optimization. *arXiv preprint arXiv:1708.01012*, 2017.

Zhu, L., Liu, Z., and Han, S. Deep leakage from gradients. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

A. Details of Experimental Setting

A.1. Server-Side Attacks

Throughout the paper, we focus on client-side attacks. We empirically show that DBCL can perfectly defend client side attacks. Here we briefly discuss server-side attacks. We assume first, the server is honest but curious, second, the server holds a subset of data, and third, the server knows the true model parameters $\mathbf{W}$, the sketched gradients, and the sketch matrix, $\mathbf{S}$. The assumptions, especially the third, make it easy to attack but hard to defend. Let $\mathbf{\Gamma}_i$ be the sketched gradient of the i-th client. The server uses $\mathbf{\Gamma}_i\mathbf{S}^T$ to approximate the true gradient; see Eqn (2). The server uses $\mathbf{\Gamma}_i\mathbf{S}^T$ for privacy inference.

A.2. Defending the Property Inference Attack (PIA) of Melis et al. (2019)

We conduct experiments on the LFW dataset by following the settings of (Melis et al., 2019). We use one server and two clients. The task of collaborative learning is the gender classification discussed in Section 6.2. The two clients collaboratively train the model on the gender classification task for $20,000$ iterations.

During the training, the attacker seeks to infer whether a single batch of photos in the victim's private dataset contain Africans or not. We consider two types of attackers. First, one client is the attacker, and one client is the victim. Section, the server is the attacker, and one client is the victim. In the latter case, the server locally holds a subset of data; the sample size is the same as the clients.

During the $1,000$th to the $20,000$th iterations of the training, the attacker estimate the gradients computed on the victim's private data. Taking the estimated gradients are input features, a random forest performs property inference attack, that is, infer whether a batch contains Africans or not. In each iteration, the attacker trains the random forest using its local data. More specifically, the attacker computes 2 gradients with property (i.e., contain Africans) and 8 gradients without property, and the attacker takes the gradients as input features for training the random forest The attacker uses a total of $190,000$ gradients for training the random forest and $19,000$ gradients for testing.

After training the random forest, the attacker uses the random forest for binary classification. The task is to infer whether a batch of the victim's (a client) private images contain Africans or not. The test data are class-imbalanced: only $3,800$ gradients are from images of Africans, whereas the rest $15,200$ are from non-Africans. We thus use AUC as the evaluation metric. Without sketching, the AUC is 1.0, which means the attacker can exactly tell whether a batch of client's images contain Africans or not. Using sketching, the AUC of client-side attack is 0.5, and the AUC of server-side attack is 0.726.

A.3. Defending the Gradient-Matching Attack of Zhu et al. (2019)

Zhu *et al.* (Zhu et al., 2019) proposed to recover the victims' data using model parameters and gradients. They seek to find a batch of images by optimization so that the resulting gradient matches the observed gradient of the victim. We use the same CNNs as (Zhu et al., 2019) to conduct experiments on the MNIST and CIFAR-10 datasets. We apply sketching to all except the output layer.

We show in Figure 4 the results of server-side attacks. Without sketching, the gradient-matching attack very well recovers the private data of the victim. However, with sketching, the gradient-matching attack completely fails. Moreover, the experiments on client-side attacks have the same results.

B. Algorithm Derivation

In this section, we derive the algorithm outlined in Section 4. In Section B.1 and B.2, we apply sketching to dense layer and convolutional layer, respectively, and derive the gradients.

B.1. Dense Layers

For simplicity, we study the case of batch size $b = 1$ for a dense layer. Let $\mathbf{x} \in \mathbb{R}^{1\times d_{\text{in}}}$ be the input, $\mathbf{W} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}$ be the parameter matrix, $\mathbf{S} \in \mathbb{R}^{d_{\text{in}}\times s}$ ($s < d_{\text{in}}$) be a sketching matrix, and

$$\mathbf{z} \;=\; \mathbf{x}\mathbf{S}(\mathbf{W}\mathbf{S})^T \in \mathbb{R}^{1\times d_{\text{out}}}$$

be the output (during training). For out-of-sample prediction, sketching is not applied, equivalently, $\mathbf{S} = \mathbf{I}_{d_{\text{in}}}$.

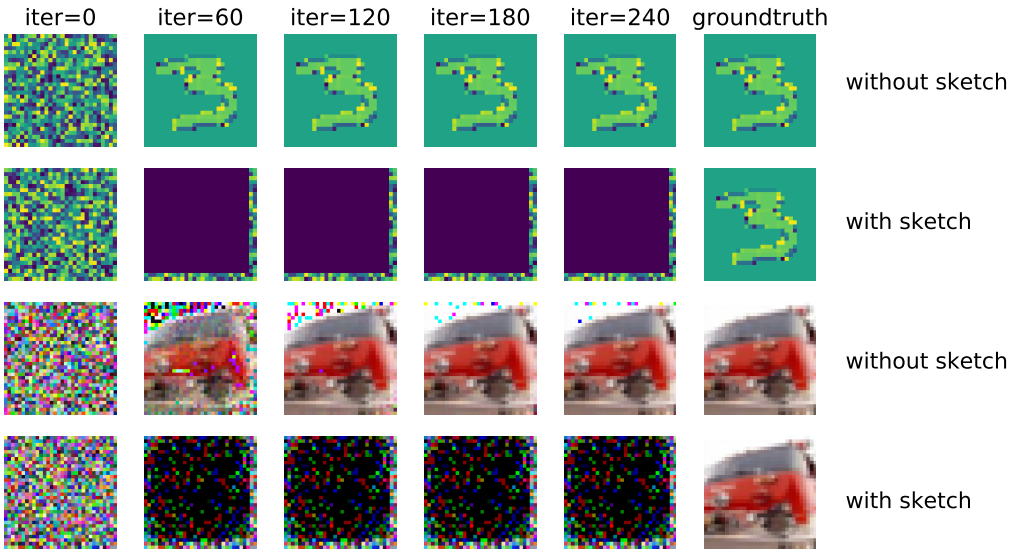


Figure 4. The images are generated by the gradient-matching attack of (Zhu et al., 2019). The attack is effective for the standard CNNs. Using sketching, the gradient-matching attack cannot recover the images.

In the following, we describe how to propagate gradient from the loss function back to $\mathbf{x}$ and $\mathbf{W}$. The dependence among the variables can be depicted as

$$\text{input} \longrightarrow \cdots \longrightarrow \underbrace{\left.\begin{matrix}\mathbf{x}\\ \mathbf{W}\end{matrix}\right\} \longrightarrow \mathbf{z}}_{\text{the studied layer}} \longrightarrow \cdots \longrightarrow \text{loss}.$$

During the backpropagation, the gradients propagated to the studied layer are

$$\mathbf{g} \triangleq \frac{\partial L}{\partial \mathbf{z}} \in \mathbb{R}^{1\times d_{\text{out}}}, \tag{6}$$

where L is some loss function. Then we further propagate the gradient from $\mathbf{z}$ to $\mathbf{x}$ and $\mathbf{W}$:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial (\mathbf{xS})} \frac{\partial (\mathbf{xS})}{\partial \mathbf{x}} = \mathbf{g}(\mathbf{WS})\mathbf{S}^T \in \mathbb{R}^{1\times d_{\text{in}}}, \tag{7}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{g}^T(\mathbf{xS})\mathbf{S}^T \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}. \tag{8}$$

We prove (8) in the following. Let $\tilde{\mathbf{x}} = \mathbf{xS} \in \mathbb{R}^{1\times s}$ and $\widetilde{\mathbf{W}} = \mathbf{WS} \in \mathbb{R}^{d_{\text{out}}\times s}$. Let $\mathbf{w}_{j:}$ and $\tilde{\mathbf{w}}_{j:}$ be the j-th row of $\mathbf{W}$ and $\widetilde{\mathbf{W}}$, respectively. It can be shown that

$$\frac{\partial L}{\partial \tilde{\mathbf{w}}_{j:}} = \sum_{l=1}^{d_{\text{out}}} \frac{\partial L}{\partial z_l} \frac{\partial z_l}{\partial \tilde{\mathbf{w}}_{j:}} = \sum_{l=1}^{d_{\text{out}}} \frac{\partial L}{\partial z_l} \frac{\partial (\tilde{\mathbf{x}}\tilde{\mathbf{w}}_{l:}^T)}{\partial \tilde{\mathbf{w}}_{j:}} = \frac{\partial L}{\partial z_j} \frac{\partial (\tilde{\mathbf{x}}\tilde{\mathbf{w}}_{j:}^T)}{\partial \tilde{\mathbf{w}}_{j:}} = g_j\, \tilde{\mathbf{x}} \in \mathbb{R}^{1\times s}.$$

Thus, $\tilde{\mathbf{w}}_{j:} = \mathbf{w}_{j:}\mathbf{S} \in \mathbb{R}^{1\times s}$; moreover, $\tilde{\mathbf{w}}_{j:}$ is independent of $\mathbf{w}_{l:}$ if $j \neq l$. It follows that

$$\frac{\partial L}{\partial \mathbf{w}_{j:}} = \frac{\partial L}{\partial \tilde{\mathbf{w}}_{j:}} \frac{\partial \tilde{\mathbf{w}}_{j:}}{\partial \mathbf{w}_{j:}} = g_j\, \tilde{\mathbf{x}}\, \mathbf{S}^T \in \mathbb{R}^{1\times d_{\text{in}}}.$$

Thus

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{g}^T\, \tilde{\mathbf{x}}\, \mathbf{S}^T = \mathbf{g}^T(\mathbf{xS})\mathbf{S}^T \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}}.$$

B.2. Extension to Convolutional Layers

Let $\mathbf{X}$ be a $d_1 \times d_2 \times d_3$ tensor and $\mathbf{K}$ be a $k_1 \times k_2 \times d_3$ kernel. The convolution $\mathbf{X} * \mathbf{K}$ outputs a $d_1 \times d_2$ matrix (assume zero-padding is used). The convolution can be equivalently written as matrix-vector multiplication in the following way.

We segment $\mathbf{X}$ to many patches of shape $k_1 \times k_2 \times d_3$ and then reshape every patch to a $d_{\text{in}} \triangleq k_1 k_2 d_3$-dimensional vector. Let $\mathbf{p}_i$ be the i-th patch (vector). Tensor $\mathbf{X}$ has $q \triangleq d_1 d_2$ such patches. Let

$$\mathbf{P} \triangleq [\mathbf{p}_1, \cdots, \mathbf{p}_q]^T \in \mathbb{R}^{q \times d_{\text{in}}}$$

be the concatenation of the patches. Let $\mathbf{w} \in \mathbb{R}^{d_{\text{in}}}$ be the vectorization of the kernel $\mathbf{K} \in \mathbb{R}^{k_1 \times k_2 \times d_3}$. The matrix-vector product, $\mathbf{z} = \mathbf{P}\mathbf{w} \in \mathbb{R}^q$, is indeed the vectorization of the convolution $\mathbf{X} * \mathbf{K}$.

In practice, we typically use multiple kernels for the convolution; let $\mathbf{W} \triangleq [\mathbf{w}_1, \cdots, \mathbf{w}_{d_{\text{out}}}]^T \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ be the concatenation of d_{out} different (vectorized) kernels. In this way, the convolution of $\mathbf{X}$ with d_{out} different kernels, which outputs a $d_1 \times d_2 \times r$ tensor, is the reshape of $\mathbf{X}\mathbf{W}^T \in \mathbb{R}^{q \times d_{\text{out}}}$.

We show in the above that tensor convolution can be equivalently expressed as matrix-matrix multiplication. Therefore, we can apply matrix sketching to convolutional layers in the same way as the dense layer. Specifically, let $\mathbf{S}$ be a $d_{\text{in}} \times s$ random sketching matrix. Then $\mathbf{X}\mathbf{S}(\mathbf{W}\mathbf{S})^T$ is an approximation to $\mathbf{X}\mathbf{W}^T$, and the backpropagation is accordingly derived using matrix differentiation.

C. Proofs

In this section, we prove Theorem 2 and Corollary 3. Theorem 2 follows from Lemmas 4 and 5. Corollary 3 follows from Lemma 6. Theorem 1 is a trivial consequence of Theorem 2.

Lemma 4. *Let $\mathbf{S}_{old}$ and $\mathbf{S}_{new}$ be independent CountSketch matrices. For any matrix $\mathbf{V}$ independent of $\mathbf{S}_{old}$ and $\mathbf{S}_{new}$, the following identity holds:*

$$\begin{aligned}
&\mathbb{E}\big\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\big\|_F^2 \\
&= \mathbb{E}\big\|\mathbf{W}_{old}\mathbf{V}^T - \mathbf{W}_{old}\mathbf{S}_{old}\mathbf{S}_{old}^T\mathbf{V}^T\big\|_F^2 + \mathbb{E}\big\|\mathbf{W}_{new}\mathbf{V}^T - \mathbf{W}_{new}\mathbf{S}_{new}\mathbf{S}_{new}^T\mathbf{V}^T\big\|_F^2,
\end{aligned}$$

where the expectation is taken w.r.t. the random sketching matrices $\mathbf{S}_{old}$ and $\mathbf{S}_{new}$.

Proof. Recall the definitions: $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$ and $\widehat{\mathbf{\Delta}} = \mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T$. Then

$$\begin{aligned}
\big\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\big\|_F^2 &= \Big\|\big(\mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\Big\|_F^2 \\
&= \Big\|\big(\mathbf{W}_{\text{old}}\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{W}_{\text{new}}\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T - (\mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}})\mathbf{V}^T\Big\|_F^2 \\
&= \Big\|\mathbf{W}_{\text{old}}\big(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}\big)\mathbf{V}^T + \mathbf{W}_{\text{new}}\big(\mathbf{I} - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T\Big\|_F^2 \\
&= \big\|\mathbf{W}_{\text{old}}\big(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}\big)\mathbf{V}^T\big\|_F^2 + \big\|\mathbf{W}_{\text{new}}\big(\mathbf{I} - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T\big\|_F^2 \\
&\quad + 2\Big\langle\mathbf{W}_{\text{old}}\big(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}\big)\mathbf{V}^T,\ \mathbf{W}_{\text{new}}\big(\mathbf{I} - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T\Big\rangle.
\end{aligned}$$

Since $\mathbb{E}[\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}] = \mathbf{0}$, $\mathbb{E}[\mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T - \mathbf{I}] = \mathbf{0}$, and $\mathbf{S}_{\text{old}}$ and $\mathbf{S}_{\text{new}}$ are independent, we have

$$\mathbb{E}\Big[\Big\langle\mathbf{W}_{\text{old}}\big(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}\big)\mathbf{V}^T,\ \mathbf{W}_{\text{new}}\big(\mathbf{I} - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T\Big\rangle\Big] = 0.$$

It follows that

$$\begin{aligned}
&\mathbb{E}\big\|\widehat{\mathbf{\Delta}}\mathbf{V}^T - \mathbf{\Delta}\mathbf{V}^T\big\|_F^2 \\
&= \mathbb{E}\big\|\mathbf{W}_{\text{old}}\big(\mathbf{S}_{\text{old}}\mathbf{S}_{\text{old}}^T - \mathbf{I}\big)\mathbf{V}^T\big\|_F^2 + \mathbb{E}\big\|\mathbf{W}_{\text{new}}\big(\mathbf{I} - \mathbf{S}_{\text{new}}\mathbf{S}_{\text{new}}^T\big)\mathbf{V}^T\big\|_F^2,
\end{aligned}$$

by which the lemma follows. □

Lemma 5. *Let* $\mathbf{S}$ *be a* $d \times s$ *CountSketch matrix. Let* $\mathbf{A} \in \mathbb{R}^{n\times d}$ *and* $\mathbf{B} \in \mathbb{R}^{m\times d}$ *be any non-random matrices. Then*

$$\mathbb{E}\big[\mathbf{A}\mathbf{S}\mathbf{S}^T\mathbf{B}^T - \mathbf{A}\mathbf{B}^T\big]^2 \;=\; \frac{1}{s}\sum_{i=1}^{n}\sum_{j=1}^{m}\Big(\sum_{k\neq l} a_{ki}^2 b_{lj}^2 + \sum_{k\neq l} a_{ki}b_{kj}a_{li}b_{lj}\Big).$$

Proof. (Pham & Pagh, 2013; Weinberger et al., 2009) showed that for any vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$,

$$\begin{aligned}\mathbb{E}\big[\mathbf{a}^T\mathbf{S}\mathbf{S}^T\mathbf{b}\big] \;&=\; \mathbf{a}^T\mathbf{b},\\ \mathbb{E}\big[\mathbf{a}^T\mathbf{S}\mathbf{S}^T\mathbf{b} - \mathbf{a}^T\mathbf{b}\big]^2 \;&=\; \frac{1}{s}\Big(\sum_{k\neq l} a_k^2 b_l^2 + \sum_{k\neq l} a_k b_k a_l b_l\Big).\end{aligned}$$

Let $\mathbf{a}_{i:} \in \mathbb{R}^d$ be the i-th row of $\mathbf{A} \in \mathbb{R}^{n\times d}$ and $\mathbf{b}_{j:} \in \mathbb{R}^d$ be the j-th column of $\mathbf{B} \in \mathbb{R}^{m\times d}$. Then,

$$\begin{aligned}\mathbb{E}\big[\mathbf{A}\mathbf{S}\mathbf{S}^T\mathbf{B}^T - \mathbf{A}\mathbf{B}^T\big]^2 \;&=\; \sum_{i=1}^{n}\sum_{j=1}^{m}\mathbb{E}\big[\mathbf{a}_{i:}^T\mathbf{S}\mathbf{S}^T\mathbf{b}_{j:} - \mathbf{a}_{i:}^T\mathbf{b}_{j:}\big]^2\\ &=\; \frac{1}{s}\sum_{i=1}^{n}\sum_{j=1}^{m}\Big(\sum_{k\neq l} a_{ik}^2 b_{jl}^2 + \sum_{k\neq l} a_{ik}b_{jk}a_{il}b_{jl}\Big),\end{aligned}$$

by which the lemma follows. □

Lemma 6. *Let* $\mathbf{S}$ *be a* $d \times s$ *CountSketch matrix. Assume that the entries of* $\mathbf{A}$ *are IID and that the entries of* $\mathbf{B}$ *are also IID. Then*

$$\mathbb{E}\big[\mathbf{A}^T\mathbf{S}\mathbf{S}^T\mathbf{B} - \mathbf{A}^T\mathbf{B}\big]^2 \;=\; \Theta\Big(\frac{d}{s}\Big)\|\mathbf{A}\mathbf{B}^T\|_F^2.$$

Proof. Assume all the entries of $\mathbf{A}$ are IID sampled from a distribution with mean μ_A and standard deviation σ_A; assume all the entries of $\mathbf{B}$ are IID sampled from a distribution with mean μ_B and standard deviation σ_B. It follows from Lemma 5 that

$$\begin{aligned}\mathbb{E}\big[\mathbf{A}\mathbf{S}\mathbf{S}^T\mathbf{B}^T - \mathbf{A}\mathbf{B}^T\big]^2 \;&=\; \frac{mn}{s}\big[(d^2-d)(\mu_A^2+\sigma_A^2)(\mu_B^2+\sigma_B^2) + (d^2-d)\mu_A^2\mu_B^2\big]\\ =\; \Theta\Big(\frac{mnd^2}{s}(\mu_A^2+\sigma_A^2)(\mu_B^2+\sigma_B^2)\Big) \;&=\; \Theta\Big(\frac{d}{s}\Big)\|\mathbf{A}\mathbf{B}^T\|_F^2.\end{aligned}$$

□

D. Decentralized Learning

Instead of relying on a central server, multiple parties can collaborate using such a peer-to-peer network as Figure 5. A client is a compute node in the graph and connected to a few neighboring nodes. Many decentralized optimization algorithms have been developed (Bianchi et al., 2013; Yuan et al., 2016; Sirb & Ye, 2016; Colin et al., 2016; Lian et al., 2017; Lan et al., 2017; Tang et al., 2018). The nodes collaborate by, for example, aggregating its neighbors' model parameters, taking a weighted average of neighbors' and its own parameters as the intermediate parameters, and then locally performing an SGD update.

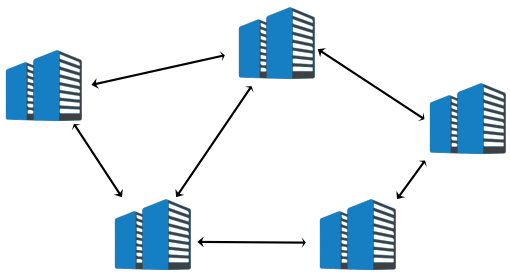


Figure 5. Decentralized learning in a peer-to-peer network.

We find that the attacks of (Melis et al., 2019; Zhu et al., 2019) can be applied to this kind of decentralized learning. Note that a node shares its model parameters with its neighbors. If a node is malicious, it can use its neighbors' gradients and model parameters to infer their data. Let a neighbor's (the victim) parameters in two consecutive rounds be $\mathbf{W}_{\text{old}}$ and $\mathbf{W}_{\text{new}}$. The

difference, $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$, is mainly the gradient evaluated on the victim's data.[7] With the model parameters $\mathbf{W}$ and updating direction $\mathbf{\Delta}$ at hand, the attacker can perform the gradient-based attacks of (Melis et al., 2019; Zhu et al., 2019).

Our DBCL can be easily applied under the decentralized setting: two neighboring compute nodes agree upon the random seeds, sketching their model parameters, and communicate the sketches. This can stop any node from knowing, $\mathbf{\Delta} = \mathbf{W}_{\text{old}} - \mathbf{W}_{\text{new}}$, i.e., the gradient of the neighbor. We will empirically study the decentralized setting in our future work.

[7] Besides the victim's gradient, $\mathbf{\Delta}$ contains the victim's neighbors' gradients, but their weights are usually lower than the victim's gradient.

E. Code of Core Algorithms

The following code implements the CountSketch and transposed CountSketch. Given matrix $\mathbf{A}$, the CountSketch outputs $\mathbf{C} = \mathbf{AS}$. Given $\mathbf{C}$, the transposed CountSketch outputs $\mathbf{B} = \mathbf{CS}^T$.

```
class Sketch():

    # Random Hashing
    # Generate random indices and random signs
    # Args:
    #    n: (integer) number of items to be hashed
    #    q: (integer) map n items to a table of s=n/q rows
    # Return:
    #    hash_idx: (q-by-s Torch Tensor) contain random integer in {0, 1, ..., s-1}
    #    rand_sgn: (n-by-1 Torch Tensor) contain random signs (+1 or -1)
    def rand_hashing(n, q):
        s = math.floor(n / q)
        t = torch.randperm(n)
        hash_idx = t[0:(s * q)].reshape((q, s))
        rand_sgn = torch.randint(0, 2, (n,)).float() * 2 - 1
        return hash_idx, rand_sgn

    # Count sketch
    # It converts m-by-n matrix to m-by-s matrix
    # Args:
    #    a: (m-by-n Torch Tensor) input matrix
    #    hash_idx: (q-by-s Torch Tensor) contain random integer in {0, 1, ..., s-1}
    #    rand_sgn: (n-by-1 Torch Tensor) contain random signs (+1 or -1)
    # Return:
    #    c: m-by-s sketch (Torch Tensor) (result of count sketch)
    def countsketch(a, hash_idx, rand_sgn):
        m, n = a.shape
        s = hash_idx.shape[1]
        c = torch.zeros([m, s], dtype=torch.float32)
        b = a.mul(rand_sgn)

        for h in range(s):
            selected = hash_idx[:, h]
            c[:, h] = torch.sum(b[:, selected], dim=1)

        return c

    # Transpose count sketch
    # The "countsketch" function converts m-by-n matrix A to m-by-s matrix C
    # This function maps C back to a m-by-n matrix B
    # Args:
    #    c: (m-by-s Torch Tensor) input matrix
    #    hash_idx: (q-by-s Torch Tensor) contain random integer in {0, 1, ..., s-1}
    #    rand_sgn: (n-by-1 Torch Tensor) contain random signs (+1 or -1)
    # Return:
    #    b: m-by-n matrix
    def transpose_countsketch(c, hash_idx, rand_sgn):
        m, s = c.shape
        n = len(rand_sgn)
        b = torch.zeros([m, n], dtype=torch.float32)
        for h in range(s):
            selected = hash_idx[:, h]
            b[:, selected] = c[:, h].reshape(m, 1)
        b = b.mul(rand_sgn)
        return b
```

The standard linear function of PyTorch computes $\mathbf{Z} = \mathbf{X}\mathbf{W}^T + \mathbf{B}$, where $\mathbf{X}$ is a batch of inputs, $\mathbf{W}$ is the weight matrix, and $\mathbf{B}$ is the bias (aka intercept). With CountSketch applied, the output becomes $\mathbf{Z} = \mathbf{X}\mathbf{S}\mathbf{S}^T\mathbf{W}^T + \mathbf{B}$. We need to implement both the forward function and the backward function. The backward function is called during backpropagation.

```
class SketchLinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias, hash_idx, rand_sgn, training=True, q=2):
        if training:
            # input_features = weight.shape[-1]

            # sketching the input and weight matrices
            # hash_idx, rand_sgn = Sketch.rand_hashing(input_features, q)
            input_sketch = Sketch.countsketch(input, hash_idx, rand_sgn)
            weight_sketch = Sketch.countsketch(weight, hash_idx, rand_sgn)
            output = input_sketch.mm(weight_sketch.t())

            ctx.save_for_backward(input_sketch, weight_sketch, bias, hash_idx, rand_sgn
                )
        else:
            output = input.mm(weight.t())

        output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        input_sketch, weight_sketch, bias, hash_idx, rand_sgn = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = grad_training = None

        if ctx.needs_input_grad[0]:
            grad_input_tmp = grad_output.mm(weight_sketch)
            grad_input = Sketch.transpose_countsketch(grad_input_tmp, hash_idx,
                rand_sgn)
        if ctx.needs_input_grad[1]:
            grad_weight_tmp = grad_output.t().mm(input_sketch)
            grad_weight = Sketch.transpose_countsketch(grad_weight_tmp, hash_idx,
                rand_sgn)
        if ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0).squeeze(0)

        return grad_input, grad_weight, grad_bias, None, None, None, None
```

The following PyTorch code applies CountSketch to the parameter matrices of a dense layer (aka linear layer or fully-connected layer).

```
class SketchLinear(nn.Module):
    def __init__(self, input_features, output_features, q=2):
        super(SketchLinear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features
        self.q = q

        self.weight = nn.Parameter(torch.Tensor(output_features, input_features))
        self.bias = nn.Parameter(torch.Tensor(output_features))
        self.register_parameter('weight', self.weight)
        self.register_parameter('bias', self.bias)

        bound = 1 / math.sqrt(input_features)
        scaling = math.sqrt(3.0)
        self.weight.data.uniform_(-bound * scaling, bound * scaling)
        self.bias.data.uniform_(-bound, bound)

    def forward(self, input, hash_idx, rand_sgn):
        return SketchLinearFunction.apply(input, self.weight, self.bias, hash_idx,
            rand_sgn, self.training, self.q)

    def extra_repr(self):
        return 'input_features={}, output_features={}, weight={}, bias={}'.format(
            self.input_features, self.output_features, self.weight, self.bias
        )
```

The following code applies CountSketch to the convolution function of PyTorch. We express convolution as matrix multiplication, and then apply CountSketch in the same way as above.

```
class SketchConvFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias, k, hash_idx, rand_sgn, training=True, q=2):
        '''
        Args:
            input: shape=(b, c0, w0, h0)
            weight: shape=(c1, c0*k*k)
            bias: shape=(c1)
            k: number of kernels

        Return:
            output: shape=(b, c1, w1, h1)

        Note:
            b: batch size
            c0: number of input channels
            c1: number of output channels
        '''
        b, c0, w0, h0 = input.shape
        c1 = weight.shape[0]
        w1 = w0 + 1 - k
        h1 = h0 + 1 - k

        # input tensor (b, c0, w0, h0) to patches (b*w1*h1, k*k*c0)
        fan_in = k * k * c0
        x = nn.functional.unfold(input, (k, k)).transpose(1, 2).reshape(b * w1 * h1,
            fan_in)

        if training:
            # sketching the input and weight matrices
            # hash_idx, rand_sgn = Sketch.rand_hashing(fan_in, q)
            x_sketch = Sketch.countsketch(x, hash_idx, rand_sgn)
            weight_sketch = Sketch.countsketch(weight, hash_idx, rand_sgn)
            z = x_sketch.matmul(weight_sketch.t())  # shape=(b*w1*h1, c1)

            # save for backprop
            shapes = torch.IntTensor([k, b, c0, w0, h0, c1, w1, h1])
            ctx.save_for_backward(x_sketch, weight_sketch, bias, shapes, hash_idx,
                rand_sgn)
        else:
            # the multiplication of x and w transpose
            z = x.matmul(weight.t())  # shape=(b*w1*h1, c1)

        # add bias
        bias_expand = bias.reshape(1, c1).expand([b * w1 * h1, c1])
        out_reshape = z + bias_expand  # shape=(b*w1*h1, c1)
        output = out_reshape.reshape(b, w1 * h1, c1).transpose(1, 2).reshape(b, c1, w1,
            h1)

        return output

    @staticmethod
    def backward(ctx, grad_output):
        x_sketch, weight_sketch, bias, shapes, hash_idx, rand_sgn = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None
        k, b, c0, w0, h0, c1, w1, h1 = shapes

        grad_output1 = grad_output.view(b, c1, -1).transpose(1, 2).reshape(b * w1 * h1,
            c1)  # shape=(b*w1*h1, c1)

        if ctx.needs_input_grad[0]:
            grad_x0 = grad_output1.matmul(weight_sketch)  # shape=(b*w1*h1, s)
```

```
59              grad_x1 = Sketch.transpose_countsketch(grad_x0, hash_idx, rand_sgn)  #
                    shape=(b*w1*h1, c0*k*k)
60              grad_x2 = grad_x1.reshape(b, w1 * h1, c0 * k * k).transpose(1, 2)
61              grad_input = nn.functional.fold(grad_x2, (w0, h0), (k, k))  # shape=(b, c0,
                    w0, h0)
62          if ctx.needs_input_grad[1]:
63              grad_w_sketch = grad_output1.t().matmul(x_sketch)  # shape=(c1, s)
64              grad_weight = Sketch.transpose_countsketch(grad_w_sketch, hash_idx,
                    rand_sgn)  # shape=(c1, c0*k*k)
65          if ctx.needs_input_grad[2]:
66              grad_bias = grad_output1.sum(0)
67
68          return grad_input, grad_weight, grad_bias, None, None, None, None, None
```

The following PyTorch code applies CountSketch to the parameter matrices of a convolutional layer.

```
class SketchConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, q=2):
        super(SketchConv, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.q = q

        self.weight = nn.Parameter(torch.Tensor(out_channels, in_channels * kernel_size
            * kernel_size))
        self.bias = nn.Parameter(torch.Tensor(out_channels))
        self.register_parameter('weight', self.weight)
        self.register_parameter('bias', self.bias)

        # uniform initialization
        scaling = math.sqrt(6.0)
        bound = 1 / math.sqrt(in_channels * kernel_size * kernel_size)
        self.weight.data.uniform_(-bound * scaling, bound * scaling)
        self.bias.data.uniform_(-bound, bound)

    def forward(self, input, hash_idx, rand_sgn):
        return SketchConvFunction.apply(input, self.weight, self.bias, self.kernel_size
            , hash_idx, rand_sgn, self.training, self.q)

    def extra_repr(self):
        return 'in_channels={}, out_channels={}, kernel_size={}, weight={}, bias={}'.
            format(
            self.in_channels, self.out_channels, self.kernel_size, self.weight, self.
                bias
        )
```