The University of Hong Kong

STAT3799 Report

Differential Privacy Meets Weight Decay: A New Optimization Approach for Deep Learning

---

Author:
Songjie Guo

Supervisor:
Prof. Yuan Cao

December 7, 2024

Abstract

Deep learning and large representative models are iterating at an unprecedented speed. However, such rapid development relies on the support of a large number of datasets. This has brought privacy concerns about the usage of datasets in deep learning to the forefront. Potential risk of data leakage under inference attacks has been proved by researchers. Differential privacy is a widely used and effective method to protect the privacy of individual information contained in datasets. It has the characteristic of achieving differential privacy according to the sensitivity of datasets, so as to enable privacy control at different levels. Based on this idea, the differentially-private stochastic gradient descent algorithm is proposed to attain differential privacy in deep learning. However, such an algorithm has convergence problems when implementing L2 regularization. To solve this issue, we propose a refined optimization method. Then we evaluate it through experiments on simulated data and real image datasets. Our method demonstrates outstanding and robust performance under different hyper parameter settings.

Acknowledgements

With the completion of this report, I would like to extend my gratitude to those who have supported me through my study at the University of Hong Kong.

First and foremost, I am grateful to Prof. Yuan Cao for their continuous support and feedback, which have not only helped me with this research but also profoundly influenced my academic journey.

Also, I would like to thank the Department of Statistics and Actuarial Science for giving me a valuable opportunity to conduct this research.

Lastly, I would also like to thank my friend Max, who have made this journey memorable and precise. Her support and encouragement have been invaluable.

Contents

List of Figures

List of Abbreviations

DP-SGD	**D**ifferentially-**P**rivate **S**tochastic **G**radient **D**escent
KL	**K**ullback–**L**eibler
MLP	**M**ulti-**L**ayer **P**erceptron
ReLU	**Re**ctified **L**inear **U**nits
SGD	**S**tochastic **G**radient **D**escent

Chapter 1

Introduction

The advances in deep learning have been tremendous in recent years, and these advancements can not be made without the support of large datasets. The essence of deep learning models is a combination of mathematical functions, which requires a massive amount of data to learn the patterns and adjust parameters. However, these datasets of texts or images may contain sensitive information about individuals. Therefore, protecting the privacy of information in datasets is of great importance when utilizing them.

Privacy is the control and protection of access to one's information. Differential privacy, proposed by Dwork, McSherry, Nissim, and Smith (2006), is a commonly recognized definition of privacy in information theory. The advantage is that it provides a quantitative method to measure the level of privacy assurance. By setting the value of the privacy budget, one can control the privacy level according to the sensitivity of the dataset.

Nasr, Shokri, and Houmansadr (2019)'s work shows that inference attacks can be applied to stochastic gradient descent (SGD) algorithm used for training in deep learning. The reason is that gradient, as a form of reflecting loss descent, contains information about the data to a certain extent. Therefore, implementing privacy preservation in the gradient is an important and effective way to ensure privacy during the training process. Differentially-private stochastic gradient descent (DP-SGD), raised by Abadi et al. (2016), is one of the most widely used techniques to achieve differential privacy in gradient.

However, DP-SGD does not detail how regularization should be applied in its algorithm. Opacus, established by Yousefpour et al. (2021), is a library to implement the idea of DP-SGD in PyTorch. Intuitively, L2 regularization in DP-SGD is realized by adding the weight decay term to the update equation in Opacus. This follows the previous idea of implementing L2 regularization in SGD in PyTorch's official library. Nevertheless, there are issues with this kind of implementation. When the values of hyperparameters are extreme, the convergence of parameters in deep neural networks is no longer guaranteed.

Our contribution in this project is summarized as the following:

1. We explore the problem of convergence with DP-SGD when hyper parameters, such as clipping threshold and weight decay, are of extreme values. We observe that when the clipping threshold is small enough, or the weight decay term is large enough, the final output of parameters trained may not relate to the local minimums at all.

2. Based on the above problems, we propose a new optimization method to implement L2 regularization in DP-SGD. Inspired by the idea of ridge regression, we successfully avoid the problem of incorrectly converging. Meanwhile, we prove that our method provides the same privacy guarantee as that in DP-SGD.

3. We evaluate the performance of our method on two datasets, MNIST and CIFAR-10. The results on these image datasets reveal that our method outperforms DP-SGD for different thresholds. Our method also ensures robust accuracy when the coefficient of L2 regularization is large.

Chapter 2

Related Work

In this chapter, we first cover the concepts of differential privacy, deep learning and regularization in Section 2.1, 2.2 and 2.3. Then, Section 2.4 introduces an algorithm that applies differential privacy in deep neural networks – Differentially Private Stochastic Gradient Descent.

2.1 Differential Privacy

Randomized algorithms are widely used in privacy related issues and cryptography. To understand differential privacy, we first review the concepts of randomized algorithm and divergence. Then we move on to introduce differential privacy and privacy loss.

Randomized algorithm It is obvious that privacy vulnerability increases if the output of a dataset is deterministic for a given input. In this situation, an attacker can always decipher the pattern through some techniques to access the private information in the data. Therefore, we need to add randomness to the outputs with randomized algorithms. Unlike deterministic algorithms, a randomized algorithm outputs a random value with a certain probability.

Definition 1 (Randomized algorithm) *A randomized algorithm $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ has domain $\mathcal{D}$ and range $\mathcal{R}$. For an input $d \in \mathcal{D}$, the output $\mathcal{M}(d)$ is a random variable with probability distribution $p(x) = \Pr[\mathcal{M}(d) = x], x \in \mathcal{R}$ and $p(x) \in \Delta(\mathcal{R})$.*

Here $\Delta(\mathcal{R})$ is a probability simplex of $\mathcal{R}$. A Probability simplex is a space with each axis representing a mutually exclusive event. Each point on the space represents a probability distribution of these events.

Definition 2 (Probability simplex) *Given a discrete set $\mathcal{S}$, its probability simplex on $\mathcal{S}$ is defined as $\Delta(\mathcal{S})$:*

$$\Delta(\mathcal{S}) = \left\{ x \in \mathbb{R}^{|\mathcal{S}|} \middle| x_i \geq 0, i = 1, 2, \cdots, |\mathcal{S}|; \sum_{i=1}^{|\mathcal{S}|} x_i = 1 \right\}$$

.

Entropy & divergence In information theory, entropy measures the uncertainty of a random variable. Kullback–Leibler (KL) divergence and max divergence are measures of the difference between two distributions. Thus, we first recall the concepts in entropy and then introduce these two divergences.

Definition 3 (Entropy) *Consider a random variable X with discrete values in a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, it has probabilities $\{p_1, p_2, \ldots, p_n\}$ where $p_i = p(x_i) = \Pr(X = x_i)$. The entropy $\mathrm{H}(x)$ is defined as*

$$\mathrm{H}(X) := -\sum_{x \in \mathcal{X}} p(x) \log p(x) = \mathbb{E}_{x \sim p}[-\log p(x)].$$

Here, $-\log p(x)$ represents the minimum encoding length of state $X = x$.

From Definition 3, we can see that the calculation of entropy requires knowing the probability distribution of each state. In practice, the true probability distribution of X may be unknown, and we can only obtain the predictive distribution of X. We denote $p(x)$ as the true distribution of X, and $q(x)$ as the predicted distribution. Cross-entropy is derived by calculating the expected value of the predicted minimum encoding length with respect to the true distribution.

Definition 4 (Cross-entropy) *Consider two distributions p, q, and they take values from the same set $\mathcal{X}$ with probabilities $p(x) = \Pr(P = x)$, $q(x) = \Pr(Q = x)$ respectively. The cross-entropy $\mathrm{H}(p, q)$ is defined as:*

$$\mathrm{H}(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x) = \mathbb{E}_{x \sim p}[-\log q(x)].$$

KL divergence, also called relative entropy, is derived from probability and information theory. It measures the difference of cross-entropy and entropy, and is calculated by $\mathrm{H}(p, q) - \mathrm{H}(p)$. A small KL divergence represents a small difference between distributions P and Q.

Definition 5 (Kullback–Leibler (KL) divergence) *Consider two random variables P and Q of the same space $\mathcal{X}$, and $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is the space from which the rows come. The probability distributions of P and Q are $p(x) = \Pr(P = x), q(x) = \Pr(Q = x), x \in \mathcal{X}$. The Kullback–Leibler (KL) divergence of P from Q is*

$$D_{KL}(P \| Q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{x \sim p}\left[\log \frac{p(x)}{q(x)}\right].$$

KL divergence measures the difference of two distributions as a whole. In terms of the maximum value of the ratio between P and Q, max divergence as the worst-case counterpart of KL divergence can be used.

Definition 6 (Max divergence) *Consider two random variables P and Q of the same space $\mathcal{X}$, and $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is the space from which the rows come. The probability distributions of P and Q are $p(x) = \Pr(P = x), q(x) = \Pr(Q = x), x \in \mathcal{X}$. The max divergence of P from Q is*

$$D_\infty(P \| Q) = \max_{x \in \mathcal{X}} \left[\log \frac{\Pr[P = x]}{\Pr[Q = x]}\right] = \max_{x \in \mathcal{X}} \left[\log \frac{p(x)}{q(x)}\right]$$

Differential privacy Differential privacy is a definition of privacy first proposed by Dwork et al. (2006). The goal of differential privacy is to ensure that the output is not sensitive to any particular record in the dataset, which means the distributions of outputs should be similar.

First, we define two datasets D and D' are adjacent if only one entry in D does not appear in D'. Consider an algorithm $\mathcal{M}$ applied to two adjacent datasets D and

D'. Differential privacy is to ensure the distributions of $\mathcal{M}(D)$ and $\mathcal{M}(D')$ are close. Therefore, we set a privacy budget ε. For any $S \subseteq \text{Range}(\mathcal{M})$, the difference between the two distributions is constrained by the:

$$\max_{x \in S}\left[\log \frac{\Pr[\mathcal{M}(D)=x]}{\Pr\left[\mathcal{M}\left(D'\right)=x\right]}\right]=\max_{S}\left[\log \frac{\Pr[\mathcal{M}(D) \in S]}{\Pr\left[\mathcal{M}\left(D'\right) \in S\right]}\right] \leq \varepsilon \tag{2.1}$$

Based on Equation 2.1, we derive the definition of differential privacy with respect to privacy budget ε and privacy leakage probability δ.

Definition 7 (Differential privacy) *Consider two adjacent datasets D and D', if a randomized algorithm $\mathcal{M}: \mathcal{D} \rightarrow \mathcal{R}$ ensures that $\Pr[\mathcal{M}(D) \in S] \leq e^{\varepsilon} \Pr\left[\mathcal{M}\left(D'\right) \in S\right]+\delta$, we say that $\mathcal{M}$ has (ε, δ)-differential privacy.*

Definition 7 allows ε-differential privacy of the algorithm $\mathcal{M}$ to be broken with probability δ. There are many randomized algorithms or mechanisms commonly used in differential privacy, including Gaussian mechanism, Laplace mechanism and Exponential mechanism. Differential privacy essentially ensures that an adversary cannot infer personal information in the dataset through queries or statistical analysis.

Another important conncept in differential privacy is privacy loss. It is used to reflect the degree of privacy leakage, and is derived from the concept of max divergence.

Definition 8 (Privacy loss) *For two adjacent datasets D and D', an algorithm $\mathcal{M}$ for a given input u generates an output v, the privacy loss is defined as:*

$$\log \frac{\Pr[\mathcal{M}(D, u)=v]}{\Pr\left[\mathcal{M}\left(D', u\right)=v\right]}.$$

2.2 Deep Learning

Deep learning relies on deep neural networks. Neural networks consist of function modules, including multilayer perceptrons (MLPs) and many non-linear activation functions, such as rectified linear units (ReLUs). The purpose of "training" is to find the most appropriate parameters in modules to connect inputs and outputs of given examples.

Loss function Feedback is effective for training, and the loss function serves as the negative feedback to the model. Loss function $\mathcal{L}(\theta)$ is defined on parameters θ. It is used to estimate the inconsistency between the model's predicted values on training examples $\{x_1, \ldots, x_N\}$ and the true outputs $\{y_1, \ldots, y_N\}$. In deep neural networks, the objective function is defined as an averaged loss function of individual examples:

$$\mathcal{L}(\theta)=\frac{1}{N} \sum_{i} \mathcal{L}\left(y_i, f\left(\theta ; x_i\right)\right).$$

The essence of the training process is to solve optimization problems based on loss functions. In other words, the aim is to find the global or local minimums:

$$\theta^*=\arg\min \mathcal{L}(\theta)=\arg\min \frac{1}{N} \sum_{i} \mathcal{L}\left(y_i, f\left(\theta ; x_i\right)\right).$$

Stochastic gradient descent Among the various optimization algorithms, the mini-batch stochastic gradient descent (SGD) algorithm is one of the most common methods. In this algorithm, we draw a small batch $B = \{x_1, \dots, x_{m'}\}$ of samples uniformly from the training set at each step. The estimate of gradient is defined as $g = \frac{1}{|B|} \sum_{x_i \in B} \nabla_\theta \mathcal{L}(y_i, f(\theta; x_i))$, and θ is updated through a descent equation:

$$\theta^{(t+1)} = \theta^{(t)} - \eta g.$$

Here η is the learning rate. It is a hyperparameter that sets the step length of each update towards the gradient descent of the loss function. The learning rate controls the speed at which the deep learning model learns.

2.3 L2 Regularization

Regularization is a set of methods used to reduce over-fitting in deep learning models. By implementing regularization, we can limit the complexity of the model to achieve a balance between complexity and performance. In this section, we introduce L2 regularization from both deep learning and statistical perspectives.

In deep learning, weight decay is applied in the descent equation to achieve L2 regularization, and λ is called "weight decay" in deep learning:

$$\theta^{(t+1)} = (1 - \eta\lambda)\theta^{(t)} - \eta\tilde{g}^{(t)}.$$

In statistics, L2 regularization is achieved through loss functions, which is known as ridge regression. Ridge regression is a modification of the loss function based on standard linear regression. Its loss function is defined as

$$J = \mathcal{L}(\theta, x) + \frac{\lambda}{2}\|\theta\|_2^2. \tag{2.2}$$

Note that, in Equation 2.2, the coefficient of L2 regularization term is set to $\frac{\lambda}{2}$ to equate the role of weight decay in deep learning.

2.4 Differentially-Private Stochastic Gradient Descent

This section describes differentially-private stochastic gradient descent (DP-SGD), an algorithm proposed by Abadi et al. (2016), toward differential privacy in deep neural networks. Inputs of this algorithm are training examples $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ and a loss function $\mathcal{L}(\theta)$. Hyperparameters include learning rate η, noise multiplier σ, clipping threshold R and weight decay λ. At each loop, a random sample is drawn from training examples with probability $q = L/N$. Then, the per-sample gradients are computed and clipped to no more than R. After that, these gradients are added to noises through a Gaussian mechanism and averaged as the gradient of the batch. Lastly, a descent equation is used to update parameters $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^{(t+1)} = (1 - \eta\lambda)\boldsymbol{\theta}^{(t)} - \eta\tilde{\mathbf{g}}^{(t)}. \tag{2.3}$$

The final output of DP-SGD is $\boldsymbol{\theta}^{(T)}$ and privacy cost (ε, δ) of the whole process through a privacy accounting method.

Chapter 3

Problem Setting

As shown in Section 2.4, DP-SGD does not clip the gradient of the weight decay term. Although such an implementation is intuitive in certain scenarios, it has potential issues. In this chapter, Section 3.1 analyses the problem of DP-SGD in weight decay, with an extreme example provided in Section 3.2.

3.1 Problems of DP-SGD

Problems of the DP-SGD algorithm occur when Equation 2.3 always holds, and it would stop updating $\boldsymbol{\theta}^{(t)}$. If assuming θ is a scalar, the solution of Equation 2.3 is $\theta^{(t)} = -\tilde{g}^{(t)}/\lambda$, which can be derived mathematically. On the other perspective, when θ is a scalar, the per-sample gradient after clipping $\overline{g}^{(t)}(x_i) = \operatorname{sgn}(g(x_i)) \cdot R$ becomes a constant if $|g^{(t)}(x_i)| > R$.

This means that when θ is a scalar, and η is small enough, $\theta^{(t)}$ has the possibility to reach a point where the descent equation always holds and stops to update. Also, the value $-\tilde{g}^{(t)}/\lambda$ is only related to the gradient or the hyperparameter λ, not the global or local minimum. In this case, $\theta^{(t)}$ may converge to irrelevant values, which are independent of the loss function $\mathcal{L}(\theta)$.

3.2 An Extreme Example

An extreme example can be used to demonstrate the problem stated in Section 3.1. For example, consider objective functions for $i = 1, \ldots, n$:

$$\mathcal{L}(\theta) = \frac{1}{2}(\theta - \theta^*)^2 \tag{3.1}$$

Here $\theta \in R$ is the trainable parameter, and θ^* is a constant. For ridge regression, the loss function in Equation 3.1 incorporates a L2 regularization term, written as

$$\mathcal{L}(\theta) = \frac{1}{2}(\theta - \theta^*)^2 + \frac{\lambda}{2}\theta^2.$$

By minimizing this loss function, $\theta = \arg\min \mathcal{L}(\theta) = \frac{1}{1+\lambda}\theta^*$. This solution is ideal as it is related to the θ^* in objective functions. However in DP-SGD, for any $\theta^* > (1 + \frac{1}{\lambda})R$, with $\theta^{(0)} = 0, \sigma = 0$ and small enough η would give iterates $\theta^{(t)}$ satisfying $\lim_{t\to\infty} \theta^{(t)} = \frac{R}{\lambda}$. The convergence of θ is not ideal in DP-SGD, as it may not relate to θ^* at all, but only depend on hyper-parameters R and λ.

In order to further demonstrate this problem, an experiment has been run on PyTorch. In the experiment, R is set to 1 and $\lambda = 0.5$. θ^* is set to 3.8. The experimental result is shown in Figure 3.1 below. The source code of this experiment is listed in Appendix B.1.

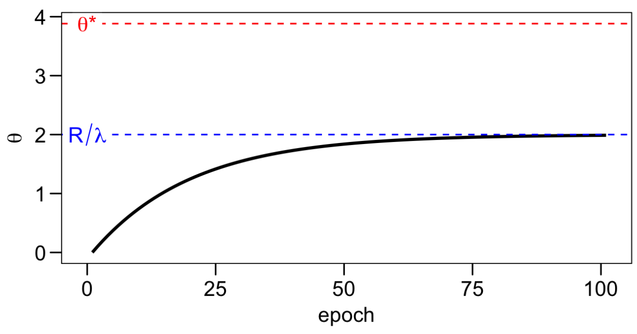


Figure 3.1: Experiment result of an extreme example of DP-SGD

Figure 3.1 shows that θ converges to $\frac{R}{\lambda} = 2$ instead of $\theta^* = 3.8$. Therefore, the convergence of parameters in DP-SGD does have problems in terms of weight decay. In order to avoid such problems, we devise a refined optimization methodology towards differentially-private deep learning with weight decay. The following Chapter 4 elaborates our method in detail.

Chapter 4

Our Method

Our method is an improved algorithm of DP-SGD in terms of problems in weight decay. By exploiting the essence of ridge regression and L2 regularization, we propose a new approach for differential privacy in deep learning. In this chapter, Section 4.1 explains our algorithm to avoid problems in DP-SGD, and Section 4.2 proves the privacy guarantee of our method.

4.1 Algorithm

Algorithm 1 outlines our method for training models in deep learning. Compared to DP-SGD, we add the L2 regularization term when computing the per-sample gradient. We change the descent equation of DP-SGD to remove the weight decay term λ so as to avoid the problem that occurs in DP-SGD. The following paragraphs discuss our improvements of DP-SGD in detail.

Algorithm 1: Our Method

```
Input: Training set $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, loss function $\mathcal{L}(\theta)$. Parameters include
       clipping threshold $R$, weight decay $\lambda$, learning rate $\eta$, noise multiplier
       $\sigma$.
for $t = [0, 1, \dots, T]$ do
    Sub-sampling
    Draw a group of random samples $I^{(t)}$ with sub-sampling $p$
    Gradient computation
    for each $i \in I^{(t)}$ do
        $\mathbf{g}_i^{(t)} = \left\{\nabla \ell\left(\boldsymbol{\theta}^{(t)}, \mathbf{x}_i, y_i\right) + \frac{\lambda}{2}\|\boldsymbol{\theta}^{(t)}\|_2^2\right\}$
        Gradient clipping
        $\overline{\mathbf{g}}_i^{(t)} \leftarrow \mathbf{g}_i^{(t)} \cdot \min(1, \frac{R}{\left\|\mathbf{g}_i^{(t)}\right\|_2})$
    Gaussian mechanism
    $\tilde{\mathbf{g}}^{(t)} \leftarrow \frac{1}{|I^{(t)}|} \sum_{i \in I^{(t)}} \left[\overline{\mathbf{g}}_i^{(t)} + \sigma R \cdot N(\mathbf{0}, \mathbf{I})\right]$ .
    Gradient descent
    $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \tilde{\mathbf{g}}^{(t)}$
Output: $\boldsymbol{\theta}^{(T)}$, as well as privacy cost $(\varepsilon, \delta)$ during this process
```

Weight decay before clipping Weight decay is a common way to implement L2 regularization in deep learning. By adding λ to the descent equation, deep learning achieves regularization when updating parameters. However, based on the problems discussed in Chapter 3, such an implementation of regularization term is not good enough. Therefore, we considered an equivalent implementation in our method. We put the equivalent penalty of weight decay $\frac{\lambda}{2}\|\boldsymbol{\theta}^{(t)}\|_2^2$ in the loss function $\mathcal{L}(\theta)$, to remove λ in the descent equation.

Descent equation By changing the descent equation to

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\tilde{\mathbf{g}}^{(t)} \tag{4.1}$$

$\boldsymbol{\theta}^{(t)}$ cannot stick to a point as that in DP-SGD, unless it reaches a local minimum where $\tilde{\mathbf{g}}^{(t)} = 0$. Therefore, the convergence of parameters in our method depend on the problem itself, not just hyperparameters.

4.2 Privacy Guarantee

In this section, we prove the privacy guarantee of our method with reference to DP-SGD by Abadi et al. (2016). For privacy loss defined in Definition 8, we consider it as a random variable A. We define the t-th moment of A as $M_A(t)$:

$$M_A(t) = \mathbb{E}_{v\sim\mathcal{M}(D,u)}[\exp(tA)] \triangleq \log \mathbb{E}_{v\sim\mathcal{M}(D,u)}[\exp(tA)].$$

Since the privacy guarantee of our method should apply to any possible output v. We define

$$M_A(t) \triangleq \max_{D,D',u} \log \mathbb{E}_{v\sim\mathcal{M}(D,u)}[\exp(tA)].$$

The random variable A satisfies the following properties:

Theorem 1 (Composability) *If an algorithm $\mathcal{M}$ is composed of algorithms $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_m$, then for any t,*

$$\mathcal{M}_A(t) \le \sum_{i=1}^{m} \mathcal{M}_{iA}(t).$$

Theorem 2 (Tail bound) *An algorithm $\mathcal{M}$ is (ε, δ)-differentially private, for any $\varepsilon > 0$ and $\delta = \min_t \exp\left(M_A(t) - t\varepsilon\right)$.*

Based on the theorems above, we can prove that Algorithm 1 satisfies differential privacy, as shown in Theorem 3 below.

Theorem 3 *There exist u and v so that given the sampling probability $q = L/N$ and the number of steps T, for any $\varepsilon < uq^2T$, our method is (ε, δ)-differentially private for any $\sigma > 0$ if we choose $\sigma \ge v\frac{q\sqrt{T\log(1/\delta)}}{\varepsilon}$.*

Therefore, our method guarantees (ε, δ)-differential privacy.

Chapter 5

Experimental Results

For the evaluation of our method, we compare its performance with DP-SGD on synthetic data and real data. For Section 5.1 and 5.2, we test on synthetic data by modeling linear regression. Further, we also evaluate the performance of our method on image datasets, including MNIST and CIFAR-10, in Section 5.3 and Section 5.4.

Inspired by the extreme examples in Section 3.2, we argue that the performance of DP-SGD suffers for extreme values of R and λ. Therefore, we mainly tune these two parameters to compare the performance of the two methods, thus demonstrating the superiority of our method.

5.1 Simple Linear Regression

First, the performance of DP-SGD and our method is derived separately using one of the simplest models, simple linear regression. Test loss is the metric to measure performance of both methods. The source code is shown in Appendix B.2.

Data generation To generate synthetic data for simple linear regression, assume

$$x \sim \mathcal{N}(0, \sigma_1^2) \tag{5.1}$$
$$\varepsilon \sim \mathcal{N}(0, \sigma_2^2) \tag{5.2}$$
$$y = wx + b + \varepsilon \tag{5.3}$$

The true w is set to 10 with $b = 0$ for simplicity. The large norm of w is set to highlight the difference in performance between DP-SGD and our method under more realistic conditions. The data generated is shown in Figure 5.1 below.

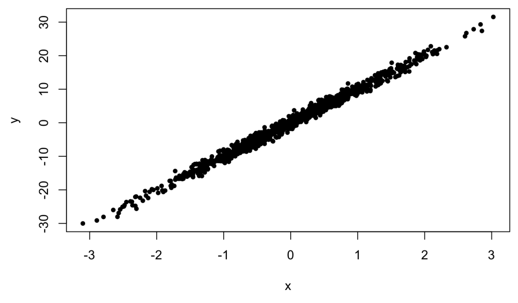


Figure 5.1: Synthetic data of linear regression

In Figure 5.1, the data generated includes a total of 1,000 examples of x and y for training and testing. We use the MSE of the test set to compare the performance of our method with DP-SGD. MSE is defined as MSE $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$.

Baseline The baseline uses a lot size L of 10, and R and λ are set to 0.1 and 0.01, respectively. As shown in Figure 5.2, we can reach a test loss of 5.93 with DP-SGD and 0.51 with our method in 100 epochs. η is set to 0.03, and $\sigma = 0.1$. In this baseline, test loss for both methods drops a lot with training and gradually approaches 0. Our method converges faster and performs better than DP-SGD.

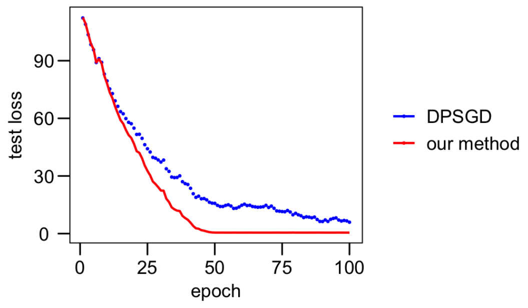


Figure 5.2: Test losses for a baseline of simple linear regression

Clipping threshold By setting clipping threshold $R = 0.01$, we can get a test loss of 84.81 with DP-SGD and 0.51 with our method, as illustrated in Figure 5.3. η is set to 0.1. When R is small enough, the test loss of DP-SGD stays at a higher level after only a partial decrease, while the performance of our method is almost unaffected after 200 epochs.

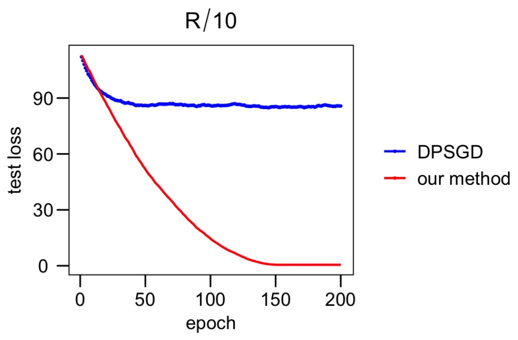


Figure 5.3: Test losses for different clipping thresholds R of simple linear regression

Weight decay By setting weight decay$\lambda = 0.1$, we can get a test loss of 79.74 with DP-SGD and 4.81 with our method, as illustrated in Figure 5.4. η is set to 0.03. The observation is that when λ is small enough, the test loss of DP-SGD fluctuates at a higher level after only a partial decrease, while our method is almost unaffected.

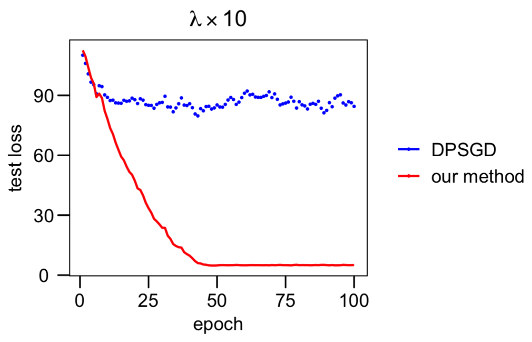


Figure 5.4: Test losses for different weight decay λ of simple linear regression

From the experiments above, we observe that:

1. The performance of DP-SGD and our method is similar when R and λ are set at proper values, or $\|w\|$ is relatively small.
2. However, when $\|w\|$ and λ are large enough, and R is small enough, DP-SGD performs poorly, but our method is almost unaffected.

5.2 Multiple Linear Regression

After the experiment of single linear regression in Section 5.1, we further explore whether the results still hold in multiple linear regression. Test loss is still used as a metric to measure the performance. The source code is shown in Appendix B.3.

Data generation To generate synthetic data for multiple linear regression, assume

$$\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}) \tag{5.4}$$

$$\varepsilon \sim \mathcal{N}(0, \sigma^2) \tag{5.5}$$

$$y = \mathbf{X}\mathbf{w} + b + \varepsilon \tag{5.6}$$

The true $\mathbf{w} = [10, 5]$ and $b = 0$. The large norm of $\mathbf{w}$ corresponds to the setting in Section 5.1.

Baseline The baseline uses the same setting as single linear regression. Lot size L is 10, and R and λ are set to 0.1 and 0.01 respectively. As shown in Figure 5.5, we can reach a test loss of 13.26 with DP-SGD and 0.63 with our method in 100 epochs. η is set to 0.03, and $\sigma = 0.1$. In this baseline, test loss for both methods gradually approaches 0 through training. Our method converges faster to lower test loss compared to DP-SGD.

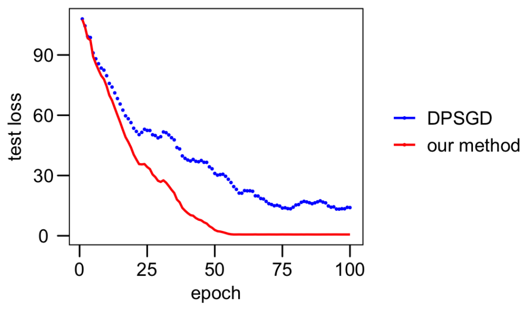


Figure 5.5: Test losses for a baseline of multiple linear regression

Clipping threshold By setting clipping threshold $R = 0.01$, we can get a test loss of 101.66 with DP-SGD and 0.63 with our method, as illustrated in Figure 5.6. The learning rate η is set to 0.1. When R is small enough, the test loss of DP-SGD maintains a higher value after only a small decrease, while the test loss of our method still goes to almost 0 after 200 epochs.

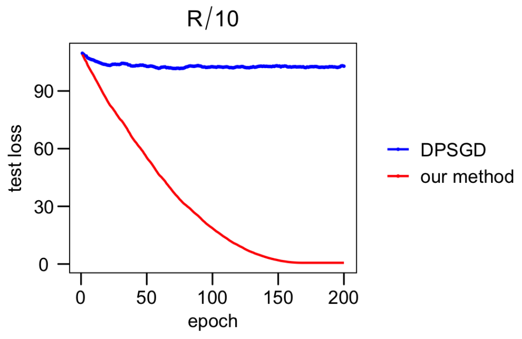


Figure 5.6: Test losses for different clipping thresholds R of multiple linear regression

Weight decay By setting weight decay$\lambda = 0.1$, we can get a test loss of 95.47 with DP-SGD and 5.96 with our method, as illustrated in Figure 5.7. The learning rate η is set to 0.03. We can observe that when λ is small enough, the test loss of DP-SGD fluctuates violently at a high level. In contrast, the performance of our method is almost unaffected.

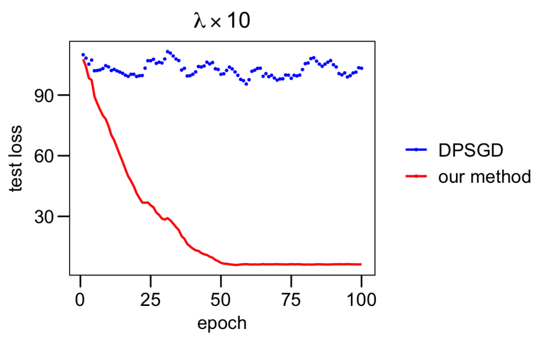


Figure 5.7: Test losses for different weight decay λ of multiple linear regression

From the experiments, we can obtain similar conclusions as simple linear regression:

1. The performance of DP-SGD and our method is similar when R and λ are set at proper values, or $\|\mathbf{w}\|$ is relatively small.
2. However, when $\|\mathbf{w}\|$ and λ are large enough, and R is small enough, DP-SGD performs poorly, but our method is almost unaffected. Opacus is a widely used library that enables training PyTorch models with differential privacy by implementing DP-SGD.

5.3 MNIST

Since we have proved that our method outperforms DP-SGD on synthetic data, we now move on to evaluate on real image dataset MNIST in this section.

In this section and Section 5.4, we apply the same model as the example in Opacus, as well as its criterion for training and testing. Opacus, built by Yousefpour et al. (2021), is an open-source PyTorch library to implement differential privacy in the training of deep neural networks. We chose this library for its fast training speed, easy application and reliable benchmarks of many datasets. Simply by adjusting the loss function and the SGD optimizer, we are able to apply our method through its framework.

Dataset The MNIST database, by Lecun, Bottou, Bengio, and Haffner (1998), is a set of gray-level images of handwritten digits from 0 to 9, with a training set of 60,000 examples and a test set of 10,000 examples. These digits have been formatted and preprocessed to be centered and size-normalized for easy training and learning pattern recognition methods.

Model To train on MNIST, we use a feed-forward neural network composed of conventional layers and fully connected layers. As shown in Figure 5.8, an image of shape $(B, 1, 28, 28)$ is first connected to two 2D conventional layers followed by ReLU and max pooling. After that, we flatten the tensor and pass it through a fully connected layer with the ReLU activation function. Lastly, another fully connected layer generates 10 outputs. The predicted digit is the one with the highest probability.

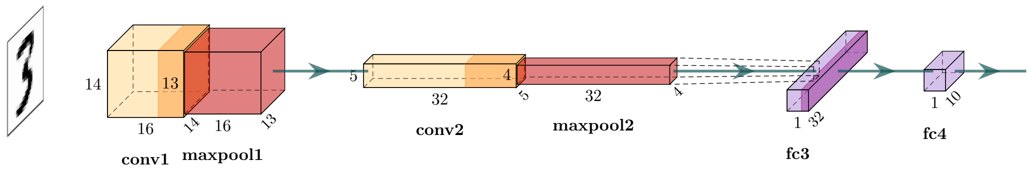


Figure 5.8: Model of MNIST

Baseline First, we set parameters at proper values to draw baselines for DP-SGD and our method, and the results are shown in Figure 5.9. After training for 50 epochs at $\delta = 10^{-5}$, our method achieves 96.89% accuracy, and DP-SGD gets 97.01% accuracy. Clipping threshold R is set to 3, weight decay λ is set to 10^{-4}, and noise multiplier σ is set to 1.3.

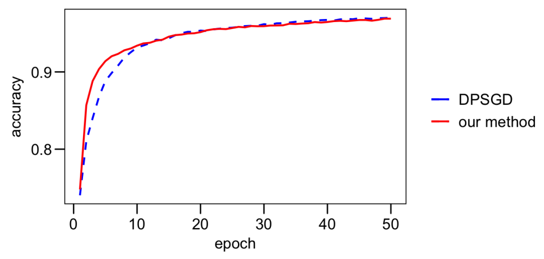


Figure 5.9: Accuracies for running a baseline using MNIST dataset

Clipping threshold To demonstrate the difference in performance of DP-SGD and our method with respect to multiple parameters, we manipulate their values individually, and the rest are fixed at baseline values. Figure 5.10 shows the accuracies of both methods on MNIST. R is set to one of $\{0.03, 0.15, 0.5, 1.5, 3\}$ at a time. We can see that although the performance of both methods deteriorates as clipping threshold R decreases, our method achieves better performance than DP-SGD for all R values.

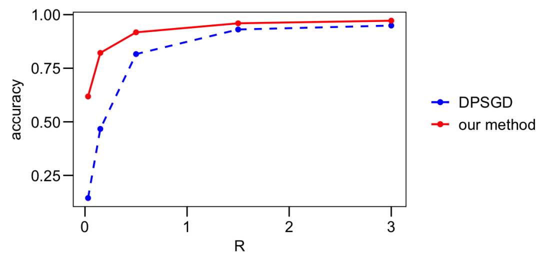
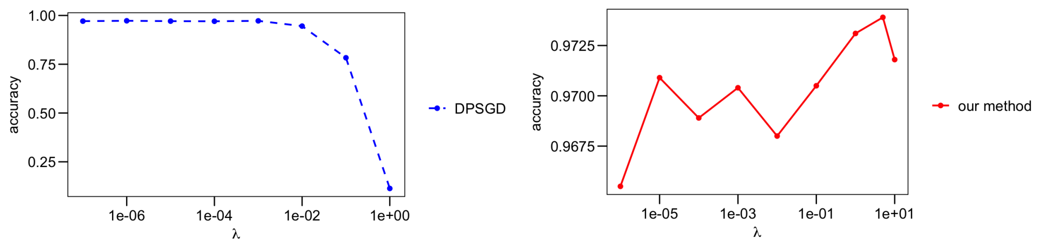


Figure 5.10: Accuracies for different clipping thresholds R using MNIST dataset

Weight decay After the discussion in the clipping threshold, we further explore the performance of these two methods under different λ. Figure 5.11shows the accuracies for different weight decay λ on MNIST with DP-SGD and our method respectively. $\lambda \in [10^{-7}, 0]$ for DP-SGD, and $\lambda \in [10^{-6}, 10]$ for our method. Other parameters are fixed at baseline values. It can be seen that when $\lambda \geq 10^{-2}$, the accuracy of DP-SGD drops sharply as λ increases. In comparison, the accuracy of our method fluctuates at a high level, around 97%.



(a) DP-SGD (b) our method

Figure 5.11: Accuracies for different weight decay λ using MNIST dataset with DP-SGD and our method respectively

In Figure 5.12, we set $\lambda \in [10^{-7}, 10]$ and put accuracies of the two methods together. We can see that our method attains the best accuracy of 97.39% when $\lambda = 5$, while DP-SGD only achieves accuracy 97.26% when $\lambda = 10^{-6}$. The reason why the best λ of our method is that the weight decay term is added to the gradient before clipping.

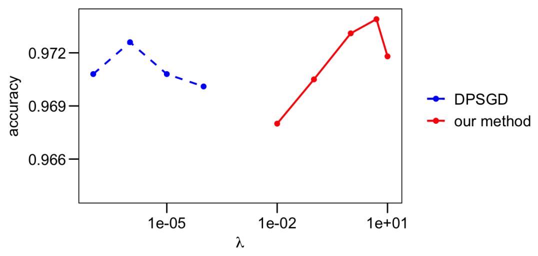


Figure 5.12: Accuracies for different weight decay λ using MNIST dataset

From the above experiments, we observe that:

1. When setting proper values for parameters, baselines are similar for DP-SGD and our method.
2. Under the same dataset, model and parameter setting, our method significantly outperforms DP-SGD when the clipping threshold is set to a relatively small value.
3. The performance of DP-SGD deteriorates when weight decay is large, while our method maintains good performance.

5.4 CIFAR-10

Dataset The CIFAR-10 Dataset by Krizhevsky (2009) is an image dataset with 60,000 32x32 color images in 10 different classes, such as airplanes and automobiles. There are five training data batches and one test data batch, each with 10,000 images.

Model The model we use is also a feed-forward neural network composed of conventional layers and fully connected layers. As shown in Figure 5.13, an image of shape $(B, 3, 32, 32)$ is first connected to four 2D conventional layers followed by ReLU and max pooling. Subsequently, the tensor is flattened and then connected to a fully connected layer that has 10 output channels. The predicted class of image is the one with the highest probability.

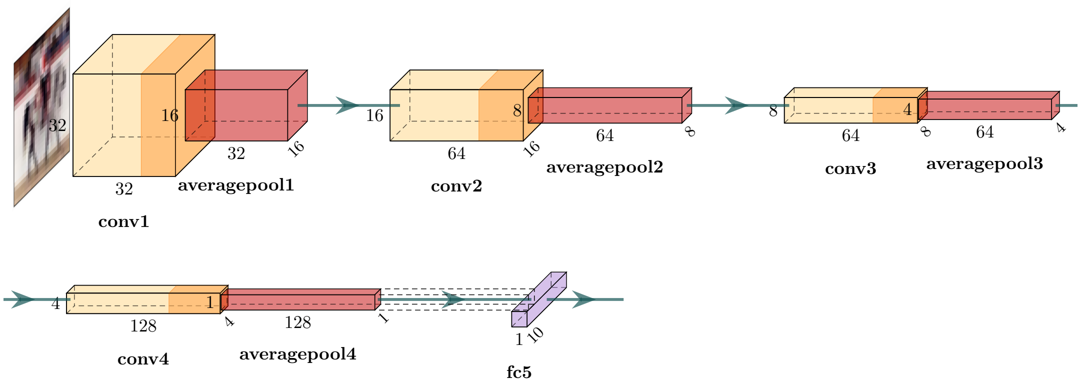


Figure 5.13: Model of CIFAR-10

Baseline In order to draw a baseline for DP-SGD and our method, we set the parameters to the appropriate values and obtain the results shown in Figure 5.14. By setting δ to 10^{-5}, we obtain similar accuracies: 65.75% for DP-SGD and 67.37% for our method. Clipping threshold R is set to 10, weight decay λ is set to 10^{-4}, and noise multiplier σ is set to 1.5.

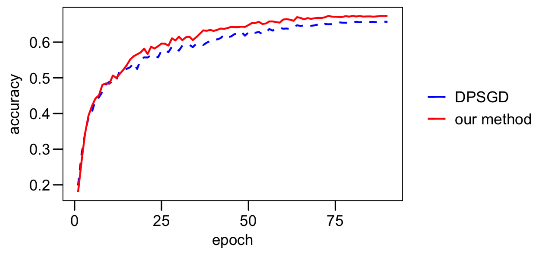


Figure 5.14: Accuracies for running a baseline using CIFAR-10 dataset

Clipping threshold To compare the performance of DP-SGD and our method for various parameters, we first experiment with different clipping thresholds. Figure 5.15 shows the accuracies of both methods on CIFAR-10. R is set to one of $\{0.2, 1, 2, 5, 10\}$ at a time, and others are fixed at baseline values. It can be seen that the performance of both methods degrades as the clipping threshold R decreases, while our method slightly outperforms DP-SGD at all R values.

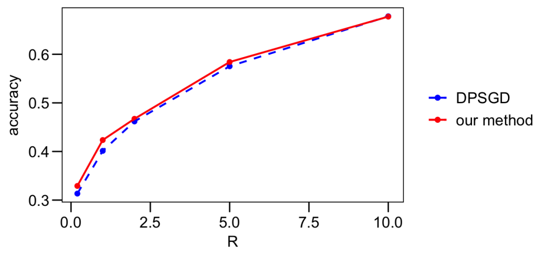
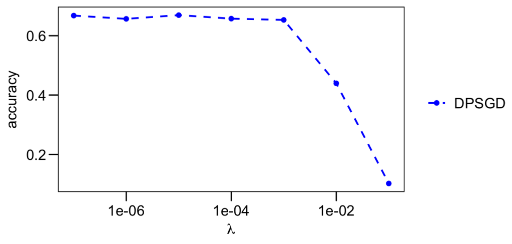
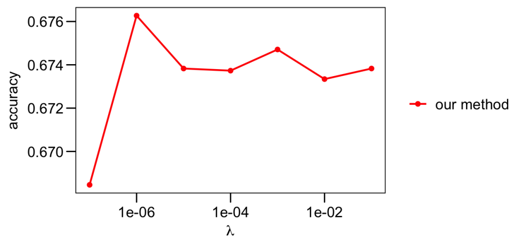


Figure 5.15: Accuracies for different clipping thresholds R using CIFAR-10 dataset

Weight decay After testing on the clipping threshold, we move on to explore the performance of these two methods under different weight decay values. Figure 5.11 shows the accuracies for different weight decay λ on CIFAR-10 with DP-SGD and our method respectively. $\lambda \in [10^{-7}, 10^{-1}]$ for both DP-SGD and our method, and other factors are fixed at baseline values. From the figure, the accuracy of DP-SGD drops sharply when $\lambda \geq 10^{-3}$, while the accuracy of our method fluctuates at a high level $\sim 67.4\%$.



(a) DP-SGD



(b) our method

Figure 5.16: Accuracies for different weight decay λ using CIFAR-10 dataset with DP-SGD and our method respectively

Figure 5.17 shows the accuracies of the two methods put together with $\lambda \in [10^{-7}, 10^{-1}]$. We can observe that our method achieves better accuracies than DP-SGD among all different weight decay values.

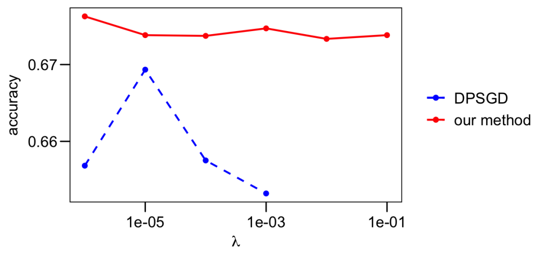


Figure 5.17: Accuracies for different weight decay λ using CIFAR-10 dataset

From the above experiments, we observe that:

1. Baselines are similar for DP-SGD and our method when we set proper values for the factors.
2. When the clipping threshold is set to a relatively small value, our method performs slightly better than DP-SGD.
3. Our method performs better than the best level of DP-SGD and does not deteriorate as the weight decay parameter increases.

Chapter 6

Conclusion

We have discovered that the convergence of DP-SGD is not robust because its weight decay term is set in the descent update equation. In order to solve the problems of DP-SGD in terms of weight decay, we proposed a new optimization approach for the training of deep neural networks with L2 regularization. From the statistical perspective, ridge regression is equivalent to the role of the weight decay term. Therefore, by implementing this idea, we add the L2 regularization term to the loss function and form a new descent equation without weight decay.

For the evaluation of our method, we first compare its performance with DP-SGD through synthetic data of linear regression models, including single linear regression and multiple linear regression. Then, we conduct some experiments on two image datasets, including MNIST and CIFAR-10, under different parameter settings for clipping threshold and weight decay. We observe that our method consistently outperforms DP-SGD across different clipping thresholds R. Additionally, our method maintains robust accuracy across different weight decay values λ, while DP-SGD performs poorly when λ is large. For further research, we would like to use larger datasets in computer vision. Other deep learning models may also be used to test the performance of our method.

Appendix A

Discussion on Noise Multiplier

In the previous discussion, we have evaluated our method for clipping thresholds R and weight decay term λ. Although theoretically, we believe that different values of noise multiplier do not lead to differences in performance between DP-SGD and our method, we conduct relevant experiments on the MNIST dataset and CIFAR-10 dataset to validate.

MNIST For the MNIST dataset, we set $\sigma \in \{1.3, 2.6, 6.5, 13, 65\}$. Others are fixed at baseline values. From Figure A.1, we can observe that the performance of both methods worsens when the noise multiplier increases, and there is no significant difference between DP-SGD and our method.

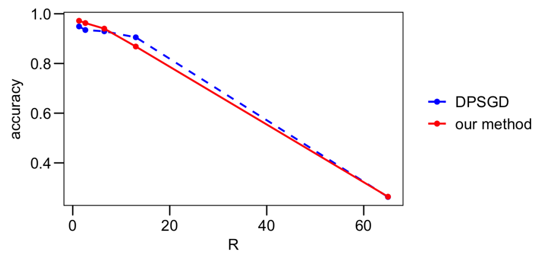


Figure A.1: Accuracies for different noise multipliers σ using MNIST dataset

CIFAR-10 For the CIFAR-10 dataset, we set $\sigma \in \{1.5, 3, 7.5, 15\}$, with other parameters fixed at baseline values. From Figure A.2, we can see that the performance of both methods deteriorates as the noise multiplier goes up, and the accuracies of DP-SGD and our method are almost the same.

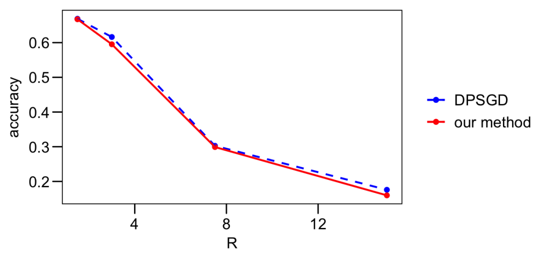


Figure A.2: Accuracies for different noise multipliers σ using CIFAR-10 dataset

In conclusion, our method and DP-SGD perform similarly with different noise multipliers σ. The performance of both methods falls off as σ goes up.

Appendix B

Code

B.1 An Extreme Example

The code below is to apply differential privacy in an extreme example in Section 3.2.

```python
import torch
torch.manual_seed(42) # set the seed

# setting
epochs = 100 # number of epochs
learning_rate = 0.1 # learning rate
momentum = 0. # momentum in SGD
weight_decay = 0.5 # weight decay in SGD
noise_multiplier = 0. # noise multiplier, set to 0 in this case
clipping_threshold = 1.0 # clipping threshold for per-sample gradient
our_method = False # whether to use our method (default: False)

lst = []
# b=0

theta_star = (1 + 1/weight_decay) * clipping_threshold + torch.rand(1).item
    ()
theta = torch.tensor(0.0, requires_grad=True)
lst.append(theta.item())

for i in range(epochs):
    loss = (theta - theta_star) ** 2
    if our_method:
        loss += learning_rate/2 * torch.norm(theta, p=2) ** 2
    loss.backward()

    with torch.no_grad():
        theta.grad = theta.grad * min(1, clipping_threshold/torch.norm(theta.
            grad, p=2))
        theta.grad = theta.grad + noise_multiplier * clipping_threshold *
            torch.randn(1).item()
        if momentum !=0:
            if i>0:
                b = momentum * b + theta.grad
            else:
                b = theta.grad
            theta.grad = b
        if not our_method:
            theta = (1 - learning_rate * weight_decay) * theta - learning_rate
                 * theta.grad
        else:
            theta = theta - learning_rate * theta.grad
        lst.append(theta.item())
    theta.requires_grad=True

print("theta:",theta.item())
print("theta*:",theta_star)
print("R/lambda:",clipping_threshold/weight_decay)

print(lst)
```

To test the performance of our method in this extreme example, change the “our_method = False” to “True” to get the code below.

```python
import torch
torch.manual_seed(42) # set the seed

# setting
epochs = 100 # number of epochs
learning_rate = 0.1 # learning rate
momentum = 0. # momentum in SGD
weight_decay = 0.5 # weight decay in SGD
noise_multiplier = 0. # noise multiplier, set to 0 in this case
clipping_threshold = 1.0 # clipping threshold for per-sample gradient
our_method = True # whether to use our method (default: False)

lst = []
# b=0

theta_star = (1 + 1/weight_decay) * clipping_threshold + torch.rand(1).item
    ()
theta = torch.tensor(0.0, requires_grad=True)
lst.append(theta.item())

for i in range(epochs):
    loss = (theta - theta_star) ** 2
    if our_method:
        loss += learning_rate/2 * torch.norm(theta, p=2) ** 2
    loss.backward()

    with torch.no_grad():
        theta.grad = theta.grad * min(1, clipping_threshold/torch.norm(theta.
            grad, p=2))
        theta.grad = theta.grad + noise_multiplier * clipping_threshold *
            torch.randn(1).item()
        if momentum !=0:
            if i>0:
                b = momentum * b + theta.grad
            else:
                b = theta.grad
            theta.grad = b
        if not our_method:
            theta = (1 - learning_rate * weight_decay) * theta - learning_rate
                 * theta.grad
        else:
            theta = theta - learning_rate * theta.grad
        lst.append(theta.item())
    theta.requires_grad=True

print("theta:",theta.item())
print("theta*:",theta_star)
print("R/lambda:",clipping_threshold/weight_decay)

print(lst)
```

B.2 Simple Linear Regression

The code below is used to get results shown in Section 5.1.

```
import torch
torch.manual_seed(42)

# setting
epochs = 100 # number of epochs
learning_rate = 0.03 # learning rate
momentum = 0. # momentum in SGD
weight_decay = 0.01 # weight decay in SGD
noise_multiplier = 0.1 # noise multiplier, set to 0 in this case
clipping_threshold = 0.1 # clipping threshold for per-sample gradient
our_method = False # whether to use our method (default: False)

# Simulate data
from sklearn.model_selection import train_test_split as tts

def synthetic_data(x_sigma, noise_sigma, true_w, true_b):
    Sy_x = torch.normal(0, x_sigma, (1000, len(true_w)))
    Sy_y = torch.matmul(Sy_x, true_w) + true_b
    Sy_y += torch.normal(0, noise_sigma, Sy_y.shape)
    Sy_x_input, Sy_x_output, Sy_y_input, Sy_y_output = tts(Sy_x, Sy_y, 0.2,
        42)
    return Sy_x_input, Sy_x_output, Sy_y_input, Sy_y_output


def load_data(Sy_x, Sy_y, batch_size, shuffle):
    from torch.utils.data import DataLoader, TensorDataset
    # for a lot/batch, shuffle = True
    # for per-sample, shuffle = False
    dataset = TensorDataset(Sy_x,Sy_y)
    dataloader = DataLoader(dataset, batch_size , shuffle)
    return dataloader

# Simple linear regression model
class SimpleLinearRegression(torch.nn.Module):
    def __init__(self, input, output):
        super(SimpleLinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input, output)
    def forward(self, x):
        slr = self.linear(x)
        return slr

def loss(slr_model, Sy_y_hat, Sy_y):
    l = torch.mean((Sy_y_hat - Sy_y.reshape(Sy_y_hat.shape)) ** 2)
    l2 = 0
    if our_method:
        for theta in slr_model.parameters():
            l2 += theta.data.norm(2) ** 2
        l2 *= weight_decay / 2
```

```python
    return l + l2


def test_loss(Sy_y_i_hat, Sy_y_i):
    l = torch.mean((Sy_y_i_hat - Sy_y_i.reshape(Sy_y_i_hat.shape)) ** 2)
    return l

# Synthetic data
x_sigma = 1
noise_sigma = 0.1
true_w = torch.tensor([10.])
true_b = torch.tensor([0.])

#
input_size = len(true_w)
output_size = len(true_b)
Sy_x_input, Sy_x_output, Sy_y_input, Sy_y_output = synthetic_data(x_sigma,
    noise_sigma, true_w, true_b)


slr_model = SimpleLinearRegression(input_size, output_size)
for theta in slr_model.parameters():
    theta.accumulated_data = []

from tqdm import tqdm
test_loss_lst = []

for epoch in tqdm(range(1, epochs + 1)):
    # train a epoch
    for Sy_x, Sy_y in load_data(Sy_x_input, Sy_y_input, 10, True):
        for theta in slr_model.parameters():
            theta.accumulated_grads = []

        for Sy_x_i, Sy_y_i in load_data(Sy_x, Sy_y, 1, False):
            Sy_y_i_hat = slr_model(Sy_x_i)
            l = loss(slr_model, Sy_y_i_hat, Sy_y_i)
            l.backward()

            sum_of_norm = 0
            for theta in slr_model.parameters():
                if theta.requires_grad:
                    sum_of_norm += theta.grad.data.norm(2).item() ** 2
            sum_of_norm = sum_of_norm ** 0.5
            clip_coef = min(clipping_threshold / (1e-8 + sum_of_norm), 1)

            for theta in slr_model.parameters():
                clipped_grad = theta.grad.data.mul(clip_coef)
                theta.accumulated_grads.append(clipped_grad)

        # Aggregate back
        for theta in slr_model.parameters():
            theta.grad = torch.mean(torch.stack(theta.accumulated_grads), dim
                =0)

        # Update and add noise
        for theta in slr_model.parameters():
            if our_method:
                theta.data = theta.data - learning_rate * theta.grad
            else:
                theta.data = (1 - learning_rate * weight_decay) * theta.data -
                    learning_rate * theta.grad
            theta.data += torch.normal(mean=0, std=noise_multiplier *
                clipping_threshold, size=theta.data.shape)
            theta.grad = None # Reset for next iteration

    test_loss = loss(slr_model, slr_model(Sy_x_output), Sy_y_output).item()
    test_loss_lst.append(test_loss)

    for theta in slr_model.parameters():
        theta.accumulated_data.append(theta.data.item())

print(test_loss_lst)

for theta in slr_model.parameters():
    print(theta.data.item())
```

B.3 Multiple Linear Regression

The code below is used to get results shown in Section 5.2.

```
import torch
torch.manual_seed(42)

# setting
epochs = 100 # number of epochs
learning_rate = 0.03 # learning rate
momentum = 0. # momentum in SGD
weight_decay = 0.01 # weight decay in SGD
noise_multiplier = 0.1 # noise multiplier, set to 0 in this case
clipping_threshold = 0.1 # clipping threshold for per-sample gradient
our_method = False # whether to use our method (default: False)

# Simulate data
from sklearn.model_selection import train_test_split as tts

def synthetic_data(x_sigma, noise_sigma, true_w, true_b):
    Sy_X = torch.normal(0, x_sigma, (1000, len(true_w)))
    Sy_y = torch.matmul(Sy_X, true_w) + true_b
    Sy_y += torch.normal(0, noise_sigma, Sy_y.shape)
    Sy_X_input, Sy_X_output, Sy_y_input, Sy_y_output = tts(Sy_X, Sy_y, 0.2,
        42)
    return Sy_X_input, Sy_X_output, Sy_y_input, Sy_y_output


def load_data(Sy_X, Sy_y, batch_size, shuffle):
    from torch.utils.data import DataLoader, TensorDataset
    # for a lot/batch, shuffle = True
    # for per-sample, shuffle = False
    dataset = TensorDataset(Sy_X,Sy_y)
    dataloader = DataLoader(dataset, batch_size , shuffle)
    return dataloader

# Multiple linear regression model
class MultipleLinearRegression(torch.nn.Module):
```

```python
    def __init__(self, input, output):
        super(MultipleLinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input, output)
    def forward(self, x):
        mlr = self.linear(x)
        return mlr

def loss(mlr_model, Sy_y_hat, Sy_y):
    l = torch.mean((Sy_y_hat - Sy_y.reshape(Sy_y_hat.shape)) ** 2)
    l2 = 0
    if our_method:
        for theta in mlr_model.parameters():
            l2 += theta.data.norm(2) ** 2
        l2 *= weight_decay / 2
    return l + l2


def test_loss(Sy_y_i_hat, Sy_y_i):
    l = torch.mean((Sy_y_i_hat - Sy_y_i.reshape(Sy_y_i_hat.shape)) ** 2)
    return l

# Synthetic data
x_sigma = 1
noise_sigma = 0.1
true_w = torch.tensor([10., 5.])
true_b = torch.tensor([0.])

#
input_size = len(true_w)
output_size = len(true_b)
Sy_X_input, Sy_X_output, Sy_y_input, Sy_y_output = synthetic_data(x_sigma,
    noise_sigma, true_w, true_b)


mlr_model = MultipleLinearRegression(input_size, output_size)
for theta in mlr_model.parameters():
    theta.accumulated_data = []

from tqdm import tqdm
test_loss_lst = []

for epoch in tqdm(range(1, epochs + 1)):
    # train a epoch
    for Sy_X, Sy_y in load_data(Sy_X_input, Sy_y_input, 10, True):
        for theta in mlr_model.parameters():
            theta.accumulated_grads = []

        for Sy_X_i, Sy_y_i in load_data(Sy_X, Sy_y, 1, False):
            Sy_y_i_hat = mlr_model(Sy_X_i)
            l = loss(mlr_model, Sy_y_i_hat, Sy_y_i)
            l.backward()

            sum_of_norm = 0
            for theta in mlr_model.parameters():
                if theta.requires_grad:
                    sum_of_norm += theta.grad.data.norm(2).item() ** 2
            sum_of_norm = sum_of_norm ** 0.5
            clip_coef = min(clipping_threshold / (1e-8 + sum_of_norm), 1)

            for theta in mlr_model.parameters():
                clipped_grad = theta.grad.data.mul(clip_coef)
                theta.accumulated_grads.append(clipped_grad)

        # Aggregate back
        for theta in mlr_model.parameters():
            theta.grad = torch.mean(torch.stack(theta.accumulated_grads), dim=0)

        # Update and add noise
        for theta in mlr_model.parameters():
            if our_method:
                theta.data = theta.data - learning_rate * theta.grad
            else:
                theta.data = (1 - learning_rate * weight_decay) * theta.data - learning_rate * theta.grad
            theta.data += torch.normal(mean=0, std=noise_multiplier * clipping_threshold, size=theta.data.shape)
            theta.grad = None # Reset for next iteration

    test_loss = loss(mlr_model, mlr_model(Sy_X_output), Sy_y_output).item()
    test_loss_lst.append(test_loss)

    for theta in mlr_model.parameters():
        theta.accumulated_data.append(theta.data.item())

print(test_loss_lst)

for theta in mlr_model.parameters():
    print(theta.data.item())
```

References

Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., & Zhang, L. (2016, October). Deep learning with differential privacy. In *Proceedings of the 2016 acm sigsac conference on computer and communications security.* ACM. doi: 10.1145/2976749.2978318

Dwork, C., McSherry, F., Nissim, K., & Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In S. Halevi & T. Rabin (Eds.), *Theory of cryptography* (pp. 265–284). Berlin, Heidelberg: Springer Berlin Heidelberg.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.. Retrieved from https://api.semanticscholar.org/CorpusID:18268744

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324. doi: 10.1109/5.726791

Nasr, M., Shokri, R., & Houmansadr, A. (2019, May). Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 ieee symposium on security and privacy (sp)* (p. 739–753). IEEE. Retrieved from http://dx.doi.org/10.1109/SP.2019.00065 doi: 10.1109/sp.2019.00065

Yousefpour, A., Shilov, I., Sablayrolles, A., Testuggine, D., Prasad, K., Malek, M., ... Mironov, I. (2021). Opacus: User-friendly differential privacy library in PyTorch. *arXiv preprint arXiv:2109.12298*.