# PyVacy: Towards Practical Differential Privacy for Deep Learning

Christopher Waites

Georgia Institute of Technology
North Avenue Northwest
Atlanta, GA 30332
May 4th, 2019

# ABSTRACT

In this work, we present an extension to the PyTorch deep learning framework which facilitates differentially private optimization. We discuss the algorithms provided by the extension and compare its contribution to that of other libraries in existence. We then go on to demonstrate the performance of the resulting private models on several statistical tasks, as well as the incurred overhead and structural changes of the resulting scripts. We find that our extension enables the construction of models which retain practical performance while incurring a minimal impact on existing, non-private training procedures.



***Keywords*** differential privacy, deep learning, software engineering


## 1 Introduction

A number of practical challenges have hindered the development of pervasive, differentially private analysis tools. Researchers associated with the U.S. Census Bureau in [1] have characterized these issues, stating they include "... the difficulty obtaining qualified personnel and a suitable computing environment, the difficulty accounting for all uses of the confidential data, the lack of release mechanisms that align with the needs of data users, the expectation on the part of data users that they will have access to micro-data, and the difficulty in setting the value of the privacy-loss parameter, $\epsilon$ (epsilon), and the lack of tools and trained individuals to verify the correctness of differential privacy implementations."

In addition to the recommendations made within [1], our work makes the assertion that the development of open-source software curtailed towards large and active communities would help in addressing many of these issues. To illustrate this, consider the task of epsilon determination. It should be clear that the determination of a standard value for epsilon, despite rules of thumb in existence, is not a tractable problem in the general case - it is an inherently human-centric issue rooted in the nature of the data and risks involved, changing from context to context. But through the increased existence of real applications, which is largely facilitated by the existence of available software, one could make more informed epsilon determinations going forward given prior demonstrations of the practical risks associated with a particular privacy threshold. Additionally, contributing to software in an open manner allows for several of the historical benefits associated with open-source contribution, namely crowdsourced verification of correctness and free empirical analysis.

So, given reason to believe that contribution of software in this manner would address the aforementioned issues surrounding differential privacy deployment, it begs the question as to which community this effort should be directed towards. We believe that one such candidate community would be the PyTorch community. A motivation for this choice is the sheer size and activity of its encompassing community. For example, when 43,000 machine learning related papers were pulled from `arXiv.org` over the preceding 6 years in March 2018, [2] found that, of the nine deep learning frameworks considered, PyTorch was mentioned second-most behind TensorFlow. It suffices to say that PyTorch is an immensely impactful platform in existence, and the support from its encompassing community could accelerate our practical understanding of the feasibility of privacy-centric technology, with the goal of making privacy a standard factor considered when conducting deep learning research.

Given this, we have elected to contribute the following: First, we expose several existing differentially private optimization algorithms in the form of custom, subclassed PyTorch optimizers. This includes the original stochastic gradient descent method presented in [3] as well as differentially private variants of other common optimization methods. We also improve upon existing methods presented within TensorFlow Privacy [4], including an improved epsilon determination procedure for their moments accountant as well as an implementation of a more recent privacy accounting method, referred to as the analytical moments accountant [5]. Finally, we construct an array of models which perform well in standard discriminative and generative tasks while adhering to reasonable privacy guarantees, making them free and open for augmentation. This work has been done with the overarching goal of minimizing the barrier to entry of existing deep learning practitioners who wish to conduct their work in a privacy-preserving manner. This is achieved by, rather than defining new abstractions with which the user must adapt to, adhering to pre-existing abstractions to the extent possible.

## 2 Related Work

TensorFlow [6] is a deep learning framework comparable to PyTorch which allows users to define computation graphs from operations performed on tensors while facilitating the automatic computation of gradients with respect to these operations. TensorFlow Privacy [4] is a recent extension upon this framework, with the goal of providing custom TensorFlow optimizers which train models in a differentially private manner. It additionally provides a privacy accountant utility, referred to as the moments accountant, which is responsible for tracking the privacy loss associated with a given optimization procedure over time. The goals of TensorFlow Privacy are closely aligned with the goals of this work, and is primarily differentiated in the communities they aim to support. Naturally, large portions of TensorFlow Privacy are simply incompatible with PyTorch due to differences in abstractions between the two platforms. Although, we have found that some of the core ideas behind TensorFlow Privacy have been useful in determining the appropriate features to include in an equivalent extension for PyTorch.

PySyft [7] is an existing framework for PyTorch which describes itself as "a library for encrypted, privacy preserving deep learning." To the best of our knowledge, of the libraries compatible which PyTorch, it is the most established and similar in nature to our work. It places an emphasis on federated learning and secure multiparty computation, with an expressed interest in support for differentially private learning. Although, in its current state, the only functionality concerning private optimization is a module relating to Private Aggregation of Teacher Ensembles [8]. In contrast, our work (alongside TensorFlow Privacy) places more emphasis on differentially private gradient descent methods and their associated variants. In practice, we believe this to be more aligned with the goal of minimizing the barrier to entry of existing deep learning practitioners, conforming to previous abstractions, and making fewer assumptions about the learning environment of the user.

## 3 Background

In this section we recall differential privacy, related definitions, and their application to deep learning. We also detail the high-level abstractions put forth by PyTorch.

### 3.1 Differential Privacy

An intuitive interpretation of differential privacy is that, given an algorithm which performs some analysis task on a subset of a database, we would like to assert the property that the choice to include or exclude any individual in the subset should make a negligible impact on the result of the algorithm. That is, the data of each entry is thought to be private since the behavior of the algorithm closely resembles the case where the entry is not included. Hence, differential privacy is a condition imposed upon an algorithm, not the contents of the database. Formally, this condition is expressed as the following, where two adjacent databases are ones in which $|D \cap D'| = 1$:

**Definition 3.1** (Differential Privacy)**.** A randomized mechanism $f : \mathcal{D} \to \mathcal{R}$ is said to have $(\epsilon, \delta)$-differential privacy if for any adjacent $D, D' \in \mathcal{D}$ and $S \subseteq \mathcal{R}$, it holds that $Pr[f(D) \in S] \leq e^{\epsilon} Pr[f(D') \in S] + \delta$.

We refer to the privacy parameter $\epsilon$ synonymously as the privacy loss.

An important property of differential privacy is its ease of composability. That is, one can easily compute the overall privacy loss associated with repeated, differentially private queries. We have the following to be true, referred to as the basic composition theorem:

**Theorem 1.** *Let $f_i : \mathcal{D} \to \mathcal{R}_i$ be $(\epsilon_i, \delta_i)$-differentially private for each $i \in [k]$. Then if $F : \mathcal{D} \to \prod_{i=1}^{k} R_i$ is defined as $F(x) = (f_1(x), \ldots, f_k(x))$, then $F$ is $(\sum_{i=1}^{k} \epsilon_i, \sum_{i=1}^{k} \delta_i)$-differentially private.*

One can achieve a tighter bound on the above via the advanced composition theorem, stated as follows:

**Theorem 2.** *For every $\epsilon > 0, \delta, \delta' > 0$, and $k \in N$, the class of $(\epsilon, \delta)$-differentially private mechanisms is $(\sqrt{2k \ln(1/\delta')} \cdot \epsilon + k\epsilon(e^{\epsilon} - 1), k\delta + \delta')$-differentially private under k-fold adaptive composition.*

Another important property of differentially private algorithms is their immunity to post-processing. This is to say that, for any data-independent transformation $g$ acting on the output of an $(\epsilon, \delta)$-differentially private algorithm $f$, $g(f(x))$ cannot incur any more privacy loss than $f$ itself.

Differential privacy is only one of several privacy definitions in existence. One such alternative definition is a generalization of differential privacy, referred to as $(\alpha, \epsilon)-$Rényi differential privacy [9], which uses the $alpha$-Rényi divergences between $f(D)$ and $f(D')$:

**Definition 3.2** (Rényi Differential Privacy)**.** A randomized mechanism $f : \mathcal{D} \rightarrow \mathcal{R}$ is said to have $(\alpha, \epsilon)$-Rényi differential privacy if for any adjacent $D, D' \in \mathcal{D}$, it holds that $D_\alpha(f(D)||f(D')) = \frac{1}{\alpha-1} \log E_{\theta \sim f(D')}[(\frac{f(D)(\theta)}{f(D')(\theta)})^\alpha] \leq \epsilon$.

Similar to differential privacy, Rényi differential privacy can be composed in the following way:

**Theorem 3.** *Let $f : \mathcal{D} \rightarrow \mathcal{R}_1$ be $(\alpha, \epsilon_1) - RDP$ and $g : \mathcal{R}_1 \times \mathcal{D} \rightarrow \mathcal{R}_2$ be $(\alpha, \epsilon_2) - RDP$. The the mechanism defined as $(X, Y)$ where $X \sim f(D)$ and $Y \sim g(X, D)$ satisfies $(\alpha, \epsilon_1 + \epsilon_2) - RDP$.*

From this given definition of RDP, one can deduce that $(\infty, \epsilon)$-RDP corresponds to the special case of $(\epsilon, 0)$-DP. More generally, [9] characterizes a relationship for converting between the two definitions, stated as follows:

**Theorem 4.** *If $f$ is an $(\alpha, \epsilon)$-RDP mechanism, it also satisfies $(\epsilon + \frac{\log 1/\delta}{\alpha-1}, \delta)$-differential privacy for any $0 < \delta < 1$.*

These primitives, namely differential privacy, Rényi differential privacy, properties of composition and postprocessing, as well as the conversion between the two definitions are essential in understanding standard differentially private optimization methods.

### 3.2 Differentially Private Deep Learning

Deep neural networks have been shown to be remarkably effective for a variety of machine learning tasks. In their standard form, they define a parameterized function composed of a sequence of layers, where each layer represents an operation to be performed on the output of the previous layer. Typically the goal associated with such models is to find the set of parameters which map a set of inputs to a set of outputs in a way which minimizes some error function.

A popular method for finding such parameters is via a process of stochastic gradient descent. When conducting stochastic gradient descent, one iteratively updates the parameters of the model by sampling an individual input-output pair from the dataset and partially applying their values to the error function so that the gradient of the error with respect to the parameters of the model can be computed. Then, one would update the parameters of the model in the direction opposite of the gradient, in turn minimizing the error function with respect to that example. Formally, if we let $\theta_0$ be the randomly initialized parameters of the model, $\theta_t$ be the parameters of the model at iteration $t$, $(x_t, y_t)$ be our sampled input-output pair, $L$ be our error function, and $\eta_t$ be the learning rate, we iteratively apply the following update rule:

$$\theta_{t+1} = \theta_t - \eta_t \nabla_{\theta_t} L(\theta_t, x_t, y_t) \tag{1}$$

Often one can achieve much faster convergence via minibatch gradient descent. Rather than calculating gradients with respect to individual examples, one uniformly samples a subset of $B$ examples without replacement, calculates the gradient with respect to each example, and applies the average of the gradients to the model. This corresponds to the following update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{B} \sum_{i=1}^{B} \nabla_{\theta_t} L(\theta_t, x_{(t,i)}, y_{(t,i)}) \tag{2}$$

To make an optimization algorithm like stochastic gradient descent differentially private, one could use a method referred to as the Gaussian mechanism. Abadi et al. in [3] detail such an algorithm, which acts as the foundation for differentially private stochastic gradient descent (DPSGD) variants. It characterizes the following update rule, where $C$ denotes the clipping parameter (an upper bound on the $l2$-norm of each gradient update), $[x]_C = x/max(1, ||x||_2/C)$ is a function which clips a given vector to have $l2$-norm at most $C$, and $\sigma$ denotes the noise multiplier (the ratio between the clipping parameter and the standard deviation of the noise applied to each gradient update):

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{B} \sum_{i=1}^{B} [\nabla_{\theta_t} L(\theta_t, x_{(t,i)}, y_{(t,i)})]_C + N(0, \sigma^2 C^2 I) \tag{3}$$

To calculate the privacy loss corresponding to $k$ executions of the above update rule, Abadi et al. in [3] detail the moments accountant as a method to report privacy loss over time. Their method yields tighter bounds on the privacy loss achieved by the algorithm than what is reported via straightforward advanced composition by composing over $RDP$ and converting to an equivalent $DP$ guarantee. Several variants of this update rule have been proposed, typically centered around the modification of how clipping and noise is applied throughout the training process (e.g. by modifying their values adaptively across iterations [10] and across layers).

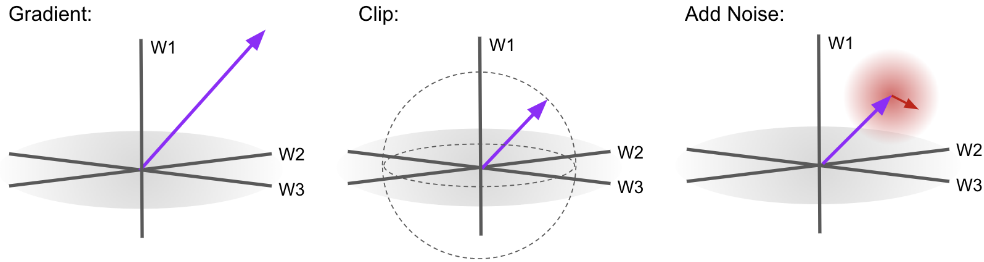



Figure 1: A visualization of the DPSGD update rule.

### 3.3 PyTorch

PyTorch is an open-source machine learning library for Python which provides support for common operations needed to train machine learning models, namely tensor arithmetic, automatic differentiation, and GPU acceleration. It imposes several abstractions which aid in the partitioning of conceptual boundaries between stages of computation while training. In its most basic form, this involves (1) a module, (2) a loss function, (3) an optimizer, and (4) a dataloader. For the purposes of this work, it is sufficient to understand these components to the following degree.

A module is an object containing one or more groups of parameters which are intended to be optimized, for example a feed-forward neural network with a group of parameters corresponding to the weights and biases of each layer in the network. A loss function characterizes some criteria for error, e.g. mean squared error or cross entropy loss. It takes in the output of some parametric computation, e.g. the output yielded from the forward pass of a neural network, compares this output to an expected value, and traverses backward through the operations which yielded the output and stores gradient updates within the parameters of each step of computation. An optimizer simply takes in a reference to a set of parameter groups and applies the stored gradient updates to their respective parameters according to some chosen method, e.g. stochastic gradient descent with momentum, Adam, AdaGrad, etc. Finally, a dataloader is responsible for batching, shuffling, and feeding examples to the model.

## 4 PyVacy

In this section, we detail the design and contributions of the presented extension. Relevant code can be found at `https://github.com/ChrisWaites/pyvacy`.

### 4.1 Differentially Private Optimizers

PyTorch exposes an appropriate context for the modification and handling of gradients in the form of PyTorch optimizers. As such, a conformant manner through which to introduce the aforementioned update rule for DPSGD would be in the form of a subclassed PyTorch optimizer, as is done in the presented extension. `pyvacy.optim.DPSGD` receives gradient updates aggregated across each minibatch and handles the logic for clipping and applying noise to the gradients. That is, it takes in $C$ and $\sigma$ and assure that the $l2$-norm of each gradient update is at most $C$, and subsequently applies Gaussian noise with standard deviation $\sigma C$ to the gradient. Each of these operations are vectorized, and as such benefit from GPU acceleration via CUDA.

```python
import torch
import torch.optim as optim

model = torch.nn.Sequential(...)

optimizer = optim.SGD(
    params=model.parameters(),
)

criterion = torch.nn.MSELoss()

dataloader = torch.utils.data.DataLoader(
    dataset=...,
    batch_size=...,
    ...
)

for epoch in range(epochs):
    for x, y in dataloader:
        y_pred = model(x)
        loss = criterion(y_pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

(a)

```python
import torch
import pyvacy.optim as optim

model = torch.nn.Sequential(...)

optimizer = optim.DPSGD(
    l2_norm_clip=...,
    noise_multiplier=...,
    batch_size=...
    params=model.parameters(),
)

acccountant = optimizer.privacy_accountant()

criterion = torch.nn.MSELoss()

dataloader = torch.utils.data.DataLoader(
    accountant,
    dataset=...,
    batch_size=...,
    ...
)

for epoch in range(epochs):
    for (x, y), eps in dataloader:
        if eps <= threshold:
            y_pred = model(x)
            loss = criterion(y_pred, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

(b)

Figure 2: Standard training procedure implemented in PyTorch, with corresponding DPSGD implementation.

An inherent constraint of PyTorch, due to the associated performance overhead, is the inability to inspect gradients with respect to each parameter on a per-example basis. That is, gradients are only available per minibatch after aggregation. Hence the update rule of the presented extension for DPSGD is more accurately described as the following, given the constraint of treating the aggregated gradient $g = \frac{1}{B}\sum_{i=1}^{B} \nabla_{\theta_t} L(\theta_t, x_{(t,i)}, y_{(t,i)})$ as immutable:

$$\theta_{t+1} = \theta_t - \eta_t([g]_C + N(0, \frac{\sigma^2 C^2}{B} I)) \tag{4}$$

Hence clipping actually occurs after aggregation, importantly maintaining the property that the gradient with respect to each individual example can still only affect the parameters of the model by at most $C$. Additionally, rather than sampling $B$ independently sampled vectors from $N(0, \sigma^2 C^2 I)$ and averaging, we equivalently apply a single vector from $N(0, \frac{\sigma^2 C^2}{B} I)$, which works around the immutability of the aggregated gradients while simultaneously yielding a performance benefit.

In addition to standard DPSGD, the extension provides support for differentially private variants of alternative optimization methods, e.g. Adam, AdaGrad, etc. This is done by simply yielding gradients in the same augmented manner to their original optimizers. This achieves, at worst, the same privacy guarantees as DPSGD via post-processing, as each modified optimization method is only ever a deterministic, data-independent function of the current and previous obfuscated gradients.

### 4.2 Privacy Accounting

Privacy accountants are responsible for defining the relationship between a given optimization method and its privacy loss achieved over time, typically utilizing some composition property. Ultimately, they wrap a function which map a set of training parameters to a set of privacy parameters which conform to some privacy definition (e.g. mapping the number of epochs, batch size, clipping parameter, noise multiplier, and target delta to epsilon in the context differential privacy). Since each optimizer contains the core logic concerning how gradients are augmented to ensure privacy, it should be the case that each optimizer contains their own separate characterization of how privacy loss is accounted for, and hence their own privacy accountant. This allows for optimizers to account for privacy according to their own potentially distinct privacy definitions, for example using RDP or zCDP rather than traditional DP.

Typically privacy accountants are able to calculate the privacy loss associated with a training procedure in advance of its actual, potentially expensive, execution. This is true within our proposed extension, as it is trivial to call the underlying

function of a given privacy accountant, iterate up to a given number of epochs, and subsequently determine how many epochs a model can be trained to achieve some target privacy loss. Although, it is a natural syntactic convenience to track privacy parameters as the model is exposed to raw data, as privacy is ultimately a characteristic of some data release mechanism. To make this natural in the context of PyTorch, recalling the previously described abstraction of a dataloader, the extension provides a convenience which allows one to provide a privacy accountant to a dataloader and yield privacy parameters alongside examples. This enables natural, dynamic, conditional behavior based off of privacy loss, as is illustrated in the main training loop of the modified script in Figure 2.

*Moments Accountant*

Abadi et al. in [3] detail the moments accountant as a privacy accounting method which yields tighter privacy bounds than straightforward advanced composition for mechanisms which subsample from a dataset. TensorFlow Privacy provides an implementation of this method which is generic enough to be reused. Although, our proposed extension provides a slightly modified version which we believe to be an improvement.

To be more concrete about the prior implementation, the moments accountant calculates the privacy loss $\epsilon$ for a DPSGD procedure with some sampling probability, noise multiplier, number of steps, and target delta. It does this by, for each $\alpha$ in a predefined set of 72 orders between 1 and 512, querying an RDP accountant with the training parameters in addition to $\alpha$ and calculates the corresponding $(\alpha, \epsilon)$-RDP achieved. It then performs the aforementioned conversion of each $(\alpha, \epsilon)$-RDP to its corresponding $(\epsilon, \delta)$-DP equivalent and yields the best (smallest) resulting $\epsilon$ achieved.

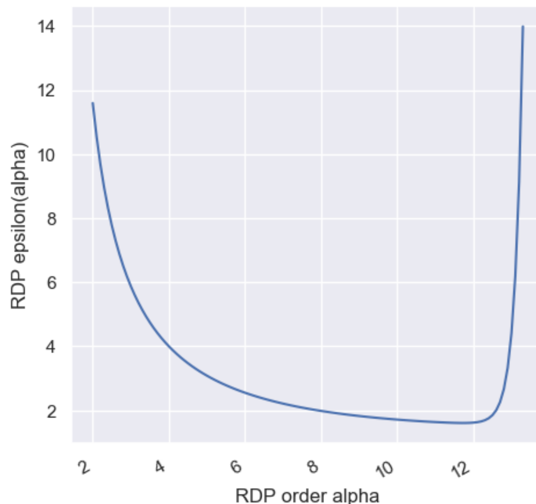



Figure 3: Epsilon as a function of the order, with $q = 0.005$, $\sigma = 1.1$, steps $= 2500$, and $\delta = 1e - 5$.

Empirically, we observe that the topology of the surface relating $\alpha$ to $\epsilon$ is unimodal and relatively sharp in growth. Given that the preselected orders are relatively granular, it is reasonable to believe that performing an explicit optimization procedure for the optimal order could yield a tighter privacy loss estimation in practice. We implemented a simple version of ternary search to find a locally optimal order assuming unimodality and was able to achieve strictly better epsilon estimates for the same number of iterations when constrained to the same domain as the original implementation. We leave for future work an investigation concerning the potential convexity of this surface.

*Analytical Moments Accountant*

Since the original detailing of the moments accountant and its subsequent implementation within TensorFlow Privacy, Wang et al. in [5] have presented an even tighter bound on the RDP privacy parameters achieved for algorithms which subsample a dataset. It presents the analytical moments accountant as a means to efficiently compose RDP across iterations and allows for easy conversion from RDP to $(\epsilon, \delta)$-DP. The core observation of their work is given as follows:

**Theorem 5.** *Given a dataset of $n$ points drawn from a domain $\mathcal{X}$ and a randomized mechanism $M$ that takes an input from $\mathcal{X}^m$ for $m \leq n$, let the randomized algorithm $M \circ \mathit{subsample}$ be defined as: (1) $\mathit{subsample}$: subsample without replacement $m$ datapoints of the dataset (sampling parameter $\gamma = m/n$), and (2) apply $M$: a randomized algorithm*

*taking the subsampled dataset as the input. For all integers $\alpha \geq 2$, if $M$ obeys $(\alpha, \epsilon(\alpha))$-RDP, then this new randomized algorithm $M \circ subsample$ obeys $(\alpha, \epsilon'(\alpha))$-RDP where,*

$$\epsilon'(\alpha) \leq \frac{1}{\alpha-1} log(1 + \gamma^2 \binom{\alpha}{2} min\{4(e^{\epsilon(2)} - 1), e^{\epsilon(2)} min\{2, (e^{\epsilon(\infty)} - 1)^2\}\} + \sum_{j=3}^{\alpha} \gamma^j \binom{\alpha}{j} e^{(j-1)\epsilon(j)} min\{2, (e^{\epsilon(\infty)} - 1)^j\})$$

An implementation of this relationship and their composition method is provided in the form of an additional privacy accountant. Although, as acknowledged within the original work, the above characterization suffers from several practical difficulties surrounding numerical stability and computability, as large constants tend to appear during the straightforward evaluation of the relationship. In our current implementation, the tighter bound is preferred for smaller orders where the above relationship is practically computable, falling back on the modified version of the moments accountant when appropriate. Future work is intended to mediate these issues via symbolic computation and further analysis of the relationship.

# 5 Empirical Evaluation

In this section, we construct several differentially private machine learning models utilizing our extension and demonstrate their performance on several statistical tasks.

## 5.1 Classification

We begin with the standard task of classification performed on the MNIST dataset [11]. Consisting of 60,000 training examples, each example is a 28 by 28 size gray-level image corresponding to an image of a handwritten digit. We use a simple feed-forward neural network with ReLU units and softmax of 10 classes, corresponding to each of the 10 digits. The model architecture is consistent with what was used in [3], and can be found in the appendix under MNIST Classifier Architecture. A training procedure was run for 60 epochs with $C = 1.0$, $\sigma = 1.1$, $\delta = 1e - 5$, and a learning rate of $0.15$, corresponding to an overall target privacy loss of $\epsilon = 3.0$. Training was executed on an Ubuntu Linux machine with a Quadro M4000 GPU, 30GB RAM, a 50GB SSD, and 8 CPU cores.

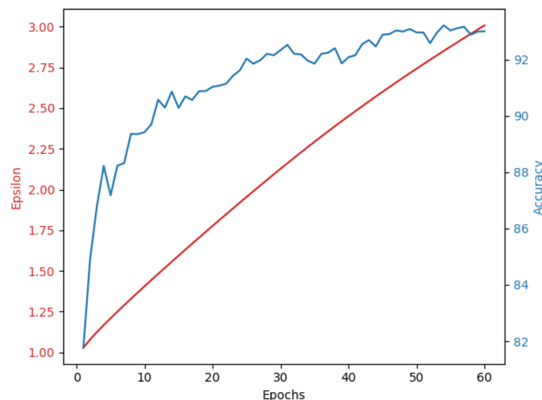



Figure 4: Learning curve for MNIST classifier with $\sigma = 1.1$ and $C = 1.0$.

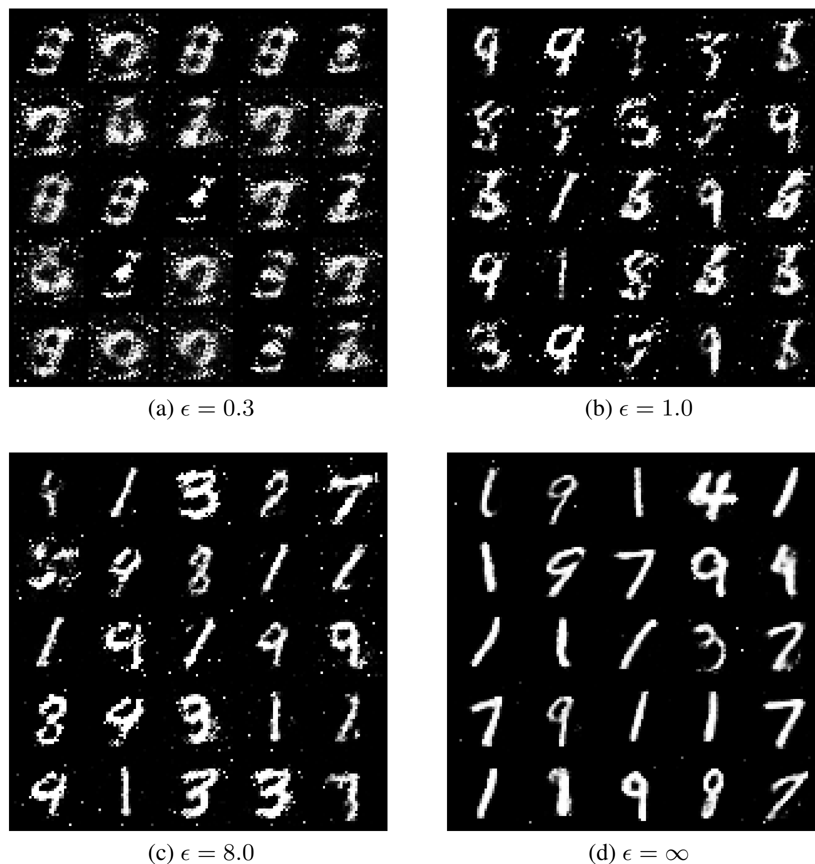
We find that the introduction of a differentially private optimization procedure achieves an out-of-sample classification accuracy of approximately 95% with $\epsilon = 3.0$, as compared to approximately 98% accuracy with the same architecture and no privacy guarantees. This was achieved with minimal structural changes to the original non-private training loop, solely involving the replacement of the original SGD optimizer with its DPSGD equivalent. To gauge the additional latency incurred by the use of the differentially private optimizer, we show the average duration (wall clock time) of an optimizer update across the training process with standard deviation error bounds.

| `optimizer.step()` | Standard Training (s) | Private Training (s) |
|---|---|---|
| No GPU | 8.93e-4 ± 1.02e-5 | 1.63e-3 ± 2.51e-5 |
| GPU | 1.94e-4 ± 1.21e-5 | 5.16e-4 ± 1.32e-5 |

We find that, associated with the introduction of a differentially private optimizer, there exists a performance deficit of approximately 2x for the same choice of the utilization of a GPU. This is to be expected, as our implementation was kept to similar performance standards of stock PyTorch optimizer implementations (proper utilization of GPU, etc.) and only included a single additional pass over the model parameters. Going forward, we intend to perform more extensive empirical evaluations of our library on larger models and datasets.

### 5.2 Data Generation

Synthetic data generation has been known to be a favorable method for facilitating the private analysis of a sensitive dataset. It allows for analysts to run unrestricted, non-private algorithms on the synthetic dataset without the need for any pre-specification of the analyses they wish to perform while still coming to nearly identical conclusions concerning the original dataset. Additionally, both the synthetic dataset and any resulting statistical results can be freely disseminated without incurring additional privacy loss. As such, finding effective, differentially private methods for synthetic data generation has been deemed an important area of research.



(a) $\epsilon = 0.3$ (b) $\epsilon = 1.0$

(c) $\epsilon = 8.0$ (d) $\epsilon = \infty$

Figure 5: Generated digits for varying levels of privacy loss.

In regard to this, several works [12, 13, 14] have proposed to approach this task via Differentially Private Generative Adversarial Networks (DPGANs). Generative Adversarial Networks [15], in their standard form, have been shown to be powerful, generic models for generating synthetic data. To make such a model differentially private, one simply augments the GAN learning process in the same manner as described before, by clipping and applying noise to gradient updates. Importantly, gradient updates with respect to the discriminator are obfuscated via DPSGD while the generator remains untouched, as it is only exposed to noise and the predictions of the discriminator, which remain private via post-processing.

We demonstrate the ability of our framework to facilitate the construction of such models by creating a standard DPGAN with two feed-forward submodules, corresponding to the generator and discriminator. The architecture in detail

can be found in the appendix under MNIST GAN Architecture. We provide the discriminator a differentially private optimizer with clipping and noise parameters corresponding to $\epsilon = 0.3, 1.0, 8.0$ and $\infty$ and a target delta $\delta = 1e-5$, approximately the inverse of the total number of examples.

We find that the resulting models are able to generate synthetic examples of reasonable quality for standard privacy loss thresholds, as shown above. Although, several commonly confronted issues concerning GANs, namely instability and mode collapse, were strictly magnified due to the addition of noise to gradient updates. Future work is intended to address these issues via a theoretical and empirical analysis of differentially private optimization on alternative GAN architectures which have been shown to be less susceptible to such issues, in particular Wasserstein-GANs [16].

## 6 Conclusion

In this work, we presented an extension to the PyTorch deep learning framework which facilitated differentially private optimization. We detailed the update rule for DPSGD and provided implementations for both it and other optimization methods while adhering to existing PyTorch abstractions. Further, we were able to show that the resulting optimizers and privacy accounts were practical in terms of their performance, and that they were able to yield scripts similar in nature to their corresponding non-private alternatives. Finally, we were able to build several models from this extension to demonstrate their retained accuracy when constrained to reasonable privacy guarantees. This work was done in hope that additional tooling in the realm of differential privacy could help address some of the difficulties currently inhibiting differential privacy deployment.

Going forward, we intend to address several pertinent issues surrounding the project. First, we aim to support additional optimization methods which stem from the initial DPSGD characterization, including methods for supporting smart clipping as detailed within [14] as well as adaptive per-iteration privacy budgeting from [10], and their respective privacy accountants. Second, we aim to address the computability issues surrounding the analytical moments accountant via the use of symbolic computation and furthered analysis of its algebraic form. Finally, we aim to further investigate the effectiveness of other machine learning models when trained in a differentially private manner, address existing instability issues surrounding GANs, and make the resulting models available via open-source.

## 7 Acknowledgements

The author thanks Rachel Cummings, Uthaipon Tantipongpipat, and Digvijay Boob for general guidance concerning the subject matter contained within the work.

## References

[1] Simson L. Garfinkel, John M. Abowd, and Sarah Powazek. Issues encountered deploying differential privacy. *CoRR*, abs/1809.02201, 2018.

[2] Andrej Karpathy, 2018.

[3] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 308–318, New York, NY, USA, 2016. ACM.

[4] Google et al. Tensorflow privacy, 2018.

[5] Yu-Xiang Wang, Borja Balle, and Shiva Kasiviswanathan. Subsampled rényi differential privacy and analytical moments accountant. *CoRR*, abs/1808.00087, 2018.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. *CoRR*, abs/1811.04017, 2018.

[8] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised knowledge transfer for deep learning from private training data. In *Proceedings of the International Conference on Learning Representations*, 2017.

[9] Ilya Mironov. Renyi differential privacy. *CoRR*, abs/1702.07476, 2017.

[10] Jaewoo Lee and Daniel Kifer. Concentrated differentially private gradient descent with adaptive per-iteration privacy budget. *CoRR*, abs/1808.09501, 2018.

[11] Yann Lecun. The mnist database of handwritten digits, 1998.

[12] Brett K. Beaulieu-Jones, Zhiwei Steven Wu, Chris Williams, and Casey S. Greene. Privacy-preserving generative deep neural networks support clinical data sharing. *bioRxiv*, 2017.

[13] Liyang Xie, Kaixiang Lin, Shu Wang, Fei Wang, and Jiayu Zhou. Differentially private generative adversarial network. *CoRR*, abs/1802.06739, 2018.

[14] Xinyang Zhang, Shouling Ji, and Ting Wang. Differentially private releasing via deep generative model. *CoRR*, abs/1801.01594, 2018.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[16] Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223. PMLR, 2017.

# A MNIST Classifier Architecture

```
Net(
  (0): Linear(in_features=784, out_features=1000, bias=True)
  (1): ReLU()
  (2): Linear(in_features=1000, out_features=10, bias=True)
  (3): Softmax()
)
```

# B GAN Architecture

```
Generator(
  (model): Sequential(
    (0): Linear(in_features=100, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): BatchNorm1d(256, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace)
    (5): Linear(in_features=256, out_features=512, bias=True)
    (6): BatchNorm1d(512, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace)
    (8): Linear(in_features=512, out_features=1024, bias=True)
    (9): BatchNorm1d(1024, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace)
    (11): Linear(in_features=1024, out_features=784, bias=True)
    (12): Tanh()
  )
)
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace)
```

```
    (4): Linear(in_features=256, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```