Robust Decentralized Differentially Private Stochastic Gradient Descent

István Hegedűs, Árpád Berta, and Márk Jelasity*
MTA-SZTE Research Group on AI
University of Szeged
Szeged, Hungary
{ihegedus, berta, jelasity}@inf.u-szeged.hu

Abstract

Stochastic gradient descent (SGD) is one of the most applied machine learning algorithms in unreliable large-scale decentralized environments. In this type of environment data privacy is a fundamental concern. The most popular way to investigate this topic is based on the framework of differential privacy. However, many important implementation details and the performance of differentially private SGD variants have not yet been completely addressed. Here, we analyze a set of distributed differentially private SGD implementations in a system, where every private data record is stored separately by an autonomous node. The examined SGD methods apply only local computations and communications contain only protected information in a differentially private manner. A key middleware service these implementations require is the single random walk service, where a single random walk is maintained in the face of different failure scenarios. First we propose a robust implementation for the decentralized single random walk service and then perform experiments to evaluate the proposed random walk service as well as the private SGD implementations. Our main conclusion here is that the proposed differentially private SGD implementations can approximate the performance of their original noise-free variants in faulty decentralized environments, provided the algorithm parameters are set properly.

Keywords: decentralized differential privacy, stochastic gradient descent, machine learning, random walks

1 Introduction

Data processing and data mining are crucial services in systems of smart appliances and the Internet of Things (IoT). Increasingly, decentralized approaches are gaining momentum, as illustrated by Cisco's ongoing fog computing initiative [1]. Compared to cloud-based solutions, an important reason is that decentralization offers better scalability by exploiting local resources and networks. Another reason is the requirement of privacy as the personal data collected and stored by smart meters, sensors, or mobile devices, is becoming ever richer and more vulnerable.

Here, we are concerned with decentralized networked systems where each device stores only a small amount of data (typically collected locally). In our study we will assume that there is a very large number (e.g. millions) of participating devices in the network. This scenario covers a wide range of systems including smart metering [2], Internet of Things platforms [3], and collaborative mobile platforms [4].

Our learning algorithm of choice is stochastic gradient descent (SGD), which visits all data records in a random order and updates a model approximation based on each record using the local gradient for that record. SGD-based algorithms are popular methods in large scale data mining [5] because of their simplicity and scalability. In our edge computing context, the simplicity of SGD is a key benefit

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 7:2 (June 2016), pp. 20-40

*Corresponding author: Hungarian Acad. Sci. and University of Szeged PO Box 652, H-6701 Szeged, Hungary, Tel: +36-62-544127, Web: http://www.inf.u-szeged.hu/~jelasity/

because all the sensitive computations can be locally performed in the network nodes. Furthermore, no synchronization, aggregation, or central collection of data is necessary.

Earlier we demonstrated the feasibility of SGD in our system model [6], but without privacy preservation.

Here, the problem we address is twofold. First, we identify the single random walk middleware service as a main component to support SGD in our system model. This service implements a random walk that allows the SGD algorithm to visit the local data records in a random order. The service should offer the illusion of a reliable single random walk while in the background it should manage the walk by possibly restarting or replicating it so as to cope with node and communication failures. Self-stabilizing leader election algorithms are perhaps the closest in that they provide the abstraction of a single entity despite faults and dynamism in arbitrary networks (see, for example, [7]). However, our problem, our system model and our priorities will be rather different.

Here, we identify three properties that are required for this abstraction. The first is that the implemented random walk has to be *agile*; that is, it should progress as quickly as possible. The second is that the implementation should be *efficient*. Therefore it should induce only a minimal extra cost to achieve robustness. For example, maintaining several replicated walks is not acceptable and ideally, the cost should be very close to that of running a single walk in a reliable system. And the third is that the random walk should be *long-lived*, meaning it should perform as many steps as possible without resetting its state.

The second problem we address is to provide an experimental evaluation of the prediction accuracy of SGD under several important design choices we identify. In our study, we will use the differential privacy [8] framework, which theoretically bounds the information leakage by adding appropriately designed noise to a query output. If we performing only secured computations, the privacy of the local data is still not guaranteed. Without differentially private mechanisms the output of any secure computation might indirectly leak information about individual data records.

The differential privacy of model fitting via optimization, and in particular SGD, was investigated earlier in several publications. Generic frameworks, like GUPT [9] and PINQ [10], have been proposed that do not willingly lend themselves to secure distributed implementations. Chaudhuri et al. propose a method based on perturbing the objective function itself, or just adding noise to the end result [11, 12]. This approach does not provide an obvious secure distributed implementation either. The method that we will build on was proposed by Song et al. [13]. There each update is made differentially private during the iterative gradient descent procedure, instead of perturbing the objective function. This approach allows for fully local gradient updates where the resulting gradient and the updated model can be made public. This prevents any uncontrolled data leakage as long as the personal computing device is not compromised.

Our contribution here is a full, practical implementation and a detailed evaluation of the feasibility of differentially private SGD in unreliable decentralized environments where each node has only one data record. More specifically, (1) we propose several variants of possible differentially private SGD implementations; (2) we propose a single random walk service that allows these SGD variants to be implemented over large decentralized systems; (3) we evaluate the random walk service over a realistic mobile phone trace; and (4) we evaluate the private SGD algorithms over several databases and parameter settings using the loss functions of support vector machines (SVM) and logistic regression.

The present study is a revised version of our previous conference publication [14], significantly extended with a novel decentralized algorithm to implement the crucial random walk service that our approach relies on, and it also includes an experimental analysis of this algorithm.

2 Background

2.1 Stochastic Gradient Descent

Classification is an important problem in machine learning. Given a data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n observations, where an object or an example is represented by a pair of a feature vector $x \in R^d$ and the corresponding class label $y \in C$, where d is the dimension of the problem and C is the domain of class labels. In the case of binary classification the number of possible class labels is two (e.g. $C = \{0, 1\}$). The problem of classification is often expressed as finding the parameters w of a function $f_w : R^d \to C$ that can correctly classify as many examples in D as possible, as well as outside D (this latter property is called generalization). In other words, we are looking for a parameter vector to optimize the objective function of the problem

$$w = \arg\min_w J(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_w(x_i), y_i) + \frac{\lambda}{2} \|w\|^2, \tag{1}$$

where the $\ell()$ is a loss function and $(\lambda/2)\|w\|^2$ is the regularization term with parameter λ. Function f_w is called the model of the data set. The regularization term helps the model to avoid overfitting the data set, thus aiding generalization. The labeled data set is often split into two non-overlapping subsets; namely a training set for optimizing the parameters w of the model and a test set for measuring the generalization performance of the optimized model.

Gradient descent (GD) is an iterative method that can find the optimum of a convex function. It is often used for optimizing the above objective function. The parameter vector w is iteratively updated using the derivative of the objective function that is computed on the whole training set

$$\begin{aligned} w_{t+1} &= w_t - \eta_t\left(\frac{\partial J}{\partial w}\right) \\ &= w_t - \eta_t\left(\lambda w + \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(f_w(x_i), y_i)\right), \end{aligned} \tag{2}$$

where η_t is the learning rate at time t that scales the size of the gradient step.

Stochastic gradient descent (SGD) is similar, only it visits each example one at a time instead of working with the entire database. It computes the gradient based on only one training sample in an iteration instead of the whole training set. For index i, the update rule becomes

$$w_{t+1} = w_t - \eta_t(\lambda w + \nabla \ell(f_w(x_i), y_i)). \tag{3}$$

SGD is more preferable on very large training sets, or in distributed applications. It has two restrictions regarding the learning rate, namely we have to have $\sum_t \eta_t^2 < \infty$ and $\sum_t \eta_t = \infty$. These turn out to be necessary conditions for convergence [15].

Here, we focus on two suitable and widely used optimization algorithms, which are *Logistic Regression* [16] and the linear *Pegasos SVM* [17]. Both have an associated loss function that we can use along with SGD to train the corresponding model. In the case of logistic regression, the optimization problem is expressed as a maximization problem, since it is more natural to think of it as maximizing the logarithm of the likelihood

$$w = \arg\max_w \frac{1}{n} \sum_{i=1}^{n} \ln P(y_i | x_i, w) - \frac{\lambda}{2} \|w\|^2, \tag{4}$$

where $y_i \in \{0, 1\}$, $P(0|x_i, w) = (1 + exp(w^T x))^{-1}$ and $P(1|x_i, w) = 1 - P(0|x_i, w)$.

Algorithm 1 Gossip Learning Framework

```
 1: (x, y) ← local training example
 2: currentModel ← initModel()
 3: loop
 4:     wait(Δ)
 5:     p ← selectPeer()
 6:     send currentModel to p
 7: end loop

 8: procedure onReceiveModel(m)
 9:     m.updateModel(x, y)
10:     currentModel ← m
11: end procedure
```

Pegasos SVM is a linear SVM solver method, which looks for the hyperplane that maximizes the margin between the instances of different classes

$$w = \arg\min_{w} \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i w^T x_i) + \frac{\lambda}{2}\|w\|^2, \tag{5}$$

where we now have $y_i \in \{-1, 1\}$.

Although we have only discussed binary classification, here we will experiment with the more general multi-class algorithms, where we have instances taken from K different classes ($C = \{0, 1, \ldots, K-1\}$). A popular approach is to learn K distinct binary classifiers [18], one for each class. In particular, for the SVM approach we optimize K hyperplanes at the same time [19]. Similarly, when using logistic regression the objective function can be readily generalized to multiple classes [18].

2.2 Distributed Machine Learning

In our system model we are given a network consisting of a large number of computational units (e.g. PCs, smart phones, tablets, wearable units, or smart meters). The members of this network can communicate with each other by message passing. A node in this network can send a message to another node whose address is known locally. We will assume that every node in this network has only one training example (but we can benefit from having more local data). The set of these isolated examples then form our machine learning database. We would like to learn a model over these instances in a fully distributed manner while preserving privacy at the same time.

The Gossip Learning Framework [6] is a possible way to learn models in this fully distributed environment. The basic idea is that in the network many models perform random walks and are updated at each node using the local example. More precisely, every node executes Algorithm 1. A node in the network first initializes a local model, then iteratively sends its local model to a randomly selected node in the network. The address of the randomly selected node is provided by a peer sampling service (e.g. the NewsCast [20] protocol). When a node receives a model, it updates it via its locally stored training example using the SGD update rule, and then stores the updated model as its local model. Using this protocol, the models stored by the nodes will converge to the same global optimum.

Our present study is based on gossip learning in the sense that we focus on SGD algorithms that are implemented through a random walk of the evolving model over the network. We will assume that this random walk itself is secure. Ideas for achieving secure random walks were recently outlined in [21] and elsewhere. Here, we focus on privacy. In order to achieve privacy, we will apply a differentially private variant of the local update step, as explained below.

2.3 Differentially Private SGD

Differential privacy [8] is concerned with the leakage of personal information due to publication of the results of a given query over a database. Even if performed securely, the result of a query can leak information about individual records, for instance, the maximum of a set of values is an individual

record in itself. Differential privacy is achieved if noise is added to the query result in such a way that the following definition is satisfied.

Definition 1 (Differential Privacy)**.** *A randomized query* $F : \mathcal{D} \mapsto \mathbb{R}^d$ *is* ϵ-differentially private *iff*

$$\forall x : e^{-\epsilon} \leq \frac{P(F(D) = x)}{P(F(D') = x)} \leq e^{\epsilon} \tag{6}$$

for all pairs of databases D and D' that differ in at most one record, where $\mathcal{D}$ is the set of possible databases.

That is, if we change one element in the database, the same output should be expected with a probability close to that over the original database. This way, one record never "matters too much", thereby limiting the information leakage as a result of the query.

A randomized query typically means adding noise to an otherwise deterministic query. This added noise is designed specifically for a given query and parameter ϵ such that the definition of $\epsilon-$differential privacy is satisfied. In more detail, to generate the additive noise we need to pick a noise distribution and the right distribution parameters. A common approach to take is to first determine the so-called *sensitivity* of the query [8,22]:

Definition 2 (Global Sensitivity)**.** *The* global L^1-sensitivity Z_F *of F is given by*

$$Z_F = \max_{D,D' \text{ differ in one record}} \|F(D) - F(D')\|_1, \tag{7}$$

where $\|\cdot\|_1$ is the L^1 norm.

The definition can be generalized by replacing the L^1 norm with a different norm. The usual norms to apply are the L^1 norm and the L^2 norm. In the case of applying the L^1 norm, the following noise distribution can be used: we need to add to all the dimensions of the output independent noise drawn from Laplace$(0, Z/\epsilon)$ (where Z is the global sensitivity of the query), which will result in $\epsilon-$differential privacy. For the L^2 norm, the noise vector should have a uniform random direction and a length drawn at random from Laplace$(0, Z/\epsilon)$. Based on the theoretical results described in [22], noise can be generated for any other norms.

Now, for one SGD update (as defined in Equation (3)) the private query we need to compute is the gradient $\nabla\ell(f_w(x_i), y_i)$. If we can guarantee that this gradient is bounded, the bound defines sensitivity directly. Having determined the sensitivity, we can then add the appropriate noise N_t to the gradient and perform the differentially private local update

$$w_{t+1} = w_t - \eta_t(\lambda w_t + \nabla\ell(f_w(x_i), y_i) + N_t). \tag{8}$$

We are then free to publish w_{t+1} and send it to the next node.

To run SGD, we require multiple queries because we need the gradients based on many learning examples multiple times. Having seen how one can protect a single update, let us mention two useful concepts from differential privacy; namely the sequential and parallel composition of queries [23].

In a sequential composition we are given a series of queries $F_i,\ i = 1, \ldots, k$. It can be proven that if all of these queries are ϵ-differentially private, then the entire sequence of these k queries will be $k \cdot \epsilon$-differentially private. Note that the queries can depend on the results of the previous queries.

However, in the special case where the k queries are executed over pairwise disjoint subsets $D_i,\ i = 1, \ldots, k$—a case we call parallel composition—the entire sequence of queries will remain ϵ-differentially private. Most importantly, in the case of SGD we have parallel composition, since updates are typically performed on a disjoint subset—in our case on a single record. Naturally the same record can be visited many times, and these updates will compose sequentially.

In general, we can think of each example as having a privacy budget of ϵ, which is spent when the given example is visited but which is not affected otherwise. This way, when each example has spent its privacy budget of ϵ, the entire SGD algorithm over the entire database will spend only ϵ due to parallel composition.

3 Private SGD Algorithm and Analysis

Now we will focus on SGD, assuming that there is a distributed implementation based on a random walk over the network. We postpone the discussion of the random walk service until Section 4, and here we treat random walk as an abstract service. Our only assumption about the implementation of the service is that the random walk is uniform; that is, it can potentially jump to any node in the network despite any technical hurdles such as NAT boxes.

The network nodes hold one training example (x, y) and they calculate the local gradient for a given model w and time t locally, and they also add the appropriate noise term N_t to achieve differential privacy based on Equation (8). They are free to publish the resulting w_{t+1} and to send it to the next node. Here, the parameter ϵ of differential privacy is a globally known constant.

3.1 Privacy Budget

The ϵ parameter is often called the privacy budget because, owing to the different compositional properties of series of queries, one can, say, decide to run one query with parameter ϵ or two sequentially composing queries with parameter $\epsilon/2$, or several parallel queries with parameter ϵ. All of these options result in an overall ϵ-differential privacy. Now, let us elaborate on the management of the privacy budget for SGD.

As mentioned in Section 2, every training example (i.e., every node) in effect has its own ϵ budget for the updates. This budget can be used in a number of different ways. One can, for instance, set a finite number of k allowed updates and use ϵ/k for each one. This means multiplying the magnitude of the noise term by k for each update. In the experimental evaluation, we study this parameter by looking at the cases of $k = 1$ and $k = 5$. We can also follow a different approach and divide ϵ into an infinite number of parts by using $\epsilon/2^t$ for update t. This way, the noise increases exponentially, but we can execute as many updates as we wish using the same example. Note, however, that SGD will not converge in this case due to the exponentially increasing noise, so this approach is practical only for a small finite number of rounds.

The above implies a deeper result: it is not possible to run SGD until we get convergence with differential privacy because we either compute just a finite number of updates (and SGD needs an unlimited number of updates for theoretical convergence) or the signal-to-noise ratio will tend to zero in the update rule, which also prevents convergence. So the best we can achieve in theory is an approximation based on a relatively small number of updates per sample. For a large number of samples, however, this may be sufficient as our experiments will demonstrate.

Let us point out a major difference between our differentially private SGD implementation and gossip learning. In our SGD implementation there is only one random walk in the entire network, while in gossip learning there are many parallel walks. However, if there are many walks in parallel, they all "burn" the privacy budget so each walk will be assigned a smaller number of updates that is inversely proportional to the number of walks. It is therefore essential to run only one walk. But, the state of the walk is public, so it is possible to continuously broadcast the latest model w_t in the network if required. The broadcast can be implemented in a distributed way (e.g. via gossip), or via publishing the latest update on a server. With public key cryptography the broadcast can be implemented securely as well. As mentioned before, it is non-trivial to run only a single random walk robustly in an unreliable system. Here, we present a service to realize this in sections 4 and 5.

As a last point connected to using the budget, let us consider the exact method of peer sampling used by our random walk. If we use uniform sampling with replacement, then the walk will take needless steps when it visits a training example that has no more budget left. To be precise, the probability that a node is not visited at all during the first n updates in a network of size n is $\exp(-1)$ according to the Poisson distribution, which is quite a large probability. Depending on the budget management option, this results in wasted bandwidth and time. This shows that the ideal random walk should use sampling

without replacement; that is, it should follow a permutation of the network, and when all nodes have been visited, it should start a new permutation until the privacy budget has been spent. This, however, is hard to realize in a decentralized manner.

3.2 Sensitivity Analysis

In our study, we focus on logistic regression and Pegasos SVM, using gradients defined in the following to equations. That is,

$$\frac{\partial J}{\partial w} = x(y - P(1|x,w)) = x(y - \frac{e^{w^T x}}{1+e^{w^T x}}) \quad y \in \{0,1\} \tag{9}$$

$$\frac{\partial J}{\partial w} = \delta(yw^T x < 1)yx, \quad y \in \{-1,1\} \tag{10}$$

In both cases the gradient is a linear function of x and its length is not larger than that of x. This immediately gives us a sensitivity of $2 \cdot \max_x \|x\|$. It is more convenient to normalize the training examples to guarantee that $\max_x \|x\| = 1$, in which case the sensitivity is 2. From now on, without loss of generality, we assume the examples are normalized this way. Actually, it is possible to achieve $\max_x \|x\| = 1$ through several different normalization methods. We will discuss these in the description part of our experiments later on. Note that we need to protect only the gradient since the update rule in Equation 8 can be computed using public information as long as $(\nabla\ell(f_w(x_i), y_i) + N_t)$ is known.

3.3 Notes on Privacy and Security

It should be stressed here that any uncorrupted node is protected by this scheme regardless of fabricated input or the security of the random walk in general. In other words, even if the random walk is compromised and a given uncorrupted node gets arbitrary input and gets queried an arbitrary number of times, the node will be protected by ϵ-differential privacy.

In this study, we focus on privacy only. Based on the comment above, this is indeed an independent problem as we can guarantee the privacy of uncompromised nodes regardless of the security of any other components of the implementation. Nevertheless, security is still vital in a complete system as without it the global output can be corrupted and vandalized. In particular, the random walk needs to be secure to maintain an unbiased sampling of the learning examples. Also, an adequate protection is required against vandalism, when adversaries or faulty nodes inject arbitrary information into the system. Again, however, the privacy of local data is completely in the hands of the local node, independently of the outside world.

4 The Single Random Walk Service

Let us now turn our attention to the implementation of the random walk service. As we mentioned before, our main goal is to propose a protocol that robustly maintains a single random walk in the system, since any extra random walks will waste the privacy budgets of the nodes without contributing to the final model. A random walk can be viewed as a mobile agent that has a state and that jumps from node to node based on local decisions at each node. The state of the walk here represents a machine learning model that is updated at each node based on local information.

Our system model is as follows. We assume there is a very large set of nodes that communicate via message passing. We also assume that a reliable transfer protocol is applied. This implies that messages are not dropped, so communication fails only if the source or target node fails before transferring the full message. At any point in time each node has a set of neighbors. The neighbor set can change over time, but nodes can send messages only to their current neighbors. Nodes can leave the network or fail at any time. In our simulations we will assume that when a node leaves the network it retains its state until it

Algorithm 2 Single Random Walk Protocol

```
 1: rwprop:                          ▹ local variable storing information about the random walk
 2: rw:                              ▹ local variable storing the state of latest visiting walk
 3: Δ:                                                                    ▹ gossip period
 4: δ:                                                                    ▹ update timeout
 5:
 6: loop                             ▹ push-pull gossip protocol to broadcast walk updates
 7:     wait(Δ)
 8:     p ← selectPeer()
 9:     send rwprop to p
10:     send pull request to p
11: end loop
12:
13: procedure onReceiveRWProps(rwprop')
14:     if (rwprop.steps < rwprop'.steps and ( rwprop.age() ≤ rwprop'.age() < δ or rwprop.age() > rwprop'.age())
         or (rwprop.steps ≥ rwprop'.steps and (rwprop.age() ≥ δ > rwprop'.age() or rwprop.age() > rwprop'.age() +δ) then
15:         rwprop ← rwprop'
16:     end if
17: end procedure
18:
19: procedure onUpdateTimeout(i)               ▹ called when rwprop.age() reaches i·δ
20:     if rwprop.rwsteps − rw.steps ≤ i then
21:         forwardRandomWalk()                                     ▹ a walk is restarted
22:     end if
23: end procedure
24:
25: procedure onReceiveRandomWalk(rw')
26:     rw'.steps ← rw'.steps+1
27:     if rw.steps < rw'.steps then
28:         rw ← rw'
29:     end if
30:     if rwprop.steps < rw.steps or rwprop.age() ≥ δ then
31:         rwprop ← new RWProp(rw.steps)
32:         forwardRandomWalk()
33:     end if
34: end procedure
```

joins the network again, but this is not a critical assumption. Messages can be delayed up to a finite time and we do not assume synchronized time.

The set of neighbors is either hard-wired, or given by other physical constraints (e.g., proximity), or set by an overlay service. Descriptions of such overlay services are widely available in the literature and fall outside the scope of our present discussion. It is not strictly required that the set of neighbors be random, but we will assume this anyway just for the sake of simplicity. If the set is not random, then implementing a random walk with a uniform stationary distribution requires additional well-proven techniques such as Metropolis-Hastings sampling or structured routing [24].

Owing to our *agility* requirement, the walk is performed in a "hot potato" style, that is, the walk moves on as soon as the local state update is performed. We should mention here that the state of the walk can be very large, possibly in the order of megabytes or more.

Let us now describe the protocol that maintains a single random walk. At any point in time, ideally there is only a single walk in the network, but—as we will see—there can be more than one walk in practice due to restarted walks based on false alarms. We will manage these walks by broadcasting a small global state about the progress of the best walk.

4.1 Broadcasting Global Updates

Every time a walk completes a new step an update is broadcast about this event. This update contains the step count of the walk in question as well as a unique id. Thus, the update is extremely small as it contains only two integer values.

The update is broadcast via a standard push-pull gossip algorithm (see Algorithm 2) that runs continuously with a period of Δ.

Every node stores only a single update locally in a variable called RWPROPS. This variable represents the local approximation of the step count of the leader random walk in the system. When a new update is received through gossip (procedure ONRECEIVERWPROPS) it has to be decided whether the new update should replace the locally stored one. Intuitively, we should replace the local update if the new update represents fresher, more up-to-date information about the best live random walk than the local RWPROPS.

In order to decide whether the update represents a live random walk, we use a timeout mechanism that is based on the age of the update. Clearly, live random walks generate events every time they make a new step. We can measure the age of these events, without global synchronization, if we accumulate the time intervals that an update spent on the nodes it visited and the total transfer time the update spent traveling over network links. This approach introduces some error into the age approximation, but our protocol is not sensitive to that error. We will revisit this issue later on.

Algorithm 2 does not contain details about the above-mentioned age accounting mechanism of the updates as it is rather technical. The approximated age is presented by the method RWPROPS.AGE that returns the current age of the update in terms of wall-clock time elapsed since the update was created.

Now, we introduce a timeout threshold δ, which represents our heuristic that if a given update is older than δ then it probably belongs to a dead random walk. The idea behind this is that if a walk does not generate updates for more than δ time then the last update it created will time out at all nodes at about the same time, clearing the way for any new walks to compete for the leadership position again.

The exact conditions for replacing the local update with the incoming one are stated in line 14. This complex formula takes into account all possible combinations of local and incoming step counts and ages, and maximizes the probability that the local update will belong to a live random walk with a maximal step count. For example, even if the local update records a larger step count, it is replaced by the incoming update if its age is smaller by at least δ, since we assume that—although the incoming update can also be outdated—the random walk recorded by the local update was probably already dead when the incoming update was created so the incoming update almost certainly represents more up-to-date information. The rest of the cases are more straightforward.

4.2 Restarting and Dropping Random Walks

In our system model—where we assumed reliable connections—a live random walk can crash only if the node that currently hosts the walk is not able to transmit the walk to any neighbor before crashing or leaving the network. With a small random walk state, the probability of this is very small, however, with the large state we have in mind it is more common for a node to crash before completing the transmission to the next node. This means that, with a positive probability, every walk can crash in each step, so the number of walks will decrease if there is no restarting mechanism in place.

First of all, to allow restarting, each node maintains a local copy of the state of the best random walk it has been visited by (variable RW) managed by method ONRECEIVERANDOMWALK. There, we store the received random walk if it has a larger step count than the previous local copy. In addition, if the random walk has a larger step count than the current best live random walk the node knows about, then the random walk is forwarded and a new update is generated. Otherwise the random walk is dropped.

As explained above, we detect the crashing of the leader random walk due to our timeout mechanism. The restarting method is based on this timeout, also taking into account our requirement that random walks should be *long-lived*.

In Algorithm 2, the event handler onUpdateTimeout takes care of restarting random walks. This handler is called when the age of the current update (rwprops) reaches $i \cdot \delta$. When $i = 1$, only the node right before the last step of the walk will try to restart the walk. If this is successful, we lose only the last step. During the next period of δ, the new walk will propagate its new updates, or the nodes will reach a timeout of 2δ. In the latter case, now both the last two nodes try to restart the walk ($i = 2$), and so on. This continues until eventually a node can successfully restart a walk. This walk will generate and broadcast new update events that will replace the timed-out updates at all the nodes.

Method forwardRandomWalk is responsible for sending the local random walk state rw to a neighbor, thus implementing one step of the walk. Here, we will not go into details about the method used in Algorithm 2. The implementation picks a neighbor and attempts to transfer the walk. This is repeated until the transfer is successful or until the forwarding is no longer necessary. The latter condition occurs if in the meantime the node gets a new update about a live walk that has a larger step count than the walk that the node is trying to forward.

4.3 Analysis

First let us give a sketch of the proof that if there are online nodes that have received at least one update before and that form a connected network then there will always be a live walk after at most a finite amount of waiting time. This is easy to see, because if there is no live walk in the network then no new events are generated, so all the nodes will eventually reach a timeout of δ. This will trigger the restart mechanism, which will eventually be successful if there is at least one online node, since eventually i will become large enough to involve all the online nodes (see method onUpdateTimeout(i)). From this point, all the online nodes will attempt a restart in every period δ until the first successful update overwrites the timed-out update.

Let us note, however, that our method does not actually guarantee that eventually there will be only a single walk. Indeed, for example, if there are two walks with the same step count that progress exactly in synchrony, making steps at exactly the same time then it is in principle possible that both of them survive indefinitely. However, we did not feel it necessary to improve our protocol to deal with this case (it would be possible with some complications) since this scenario has a very low probability. Also, if symmetry is broken then there is a positive probability that the walk with the smaller step count will hit a node that has a fresh-enough update about the walk with the larger step count, which will eventually end the walk. Instead, we opted for keeping the protocol simple and we prove experimentally that the number of concurrent walks is close to one in practice.

Let us now consider the cost of the protocol. The push-pull broadcast involves very small messages of a few dozen bytes that generate a negligible traffic on a link even if Δ is small (will will use $\Delta = 100$ ms in our tests). At the same time, push-pull broadcast results in an expected convergence time of $O(\Delta \log N)$ (where N is the network size) if peer selection is random [25]. This, considering that Δ is small, results in a reasonably fast broadcast process. To give an illustration, if the random walk has a state of 1 MB, and we set a bandwidth limit of 100 kbit/s for our application then it takes over a minute to make one step. This is about an order of magnitude more time than the broadcast convergence time in a typical network.

The random walk itself induces little traffic overall, given that we maintain just a single walk that visits any given node very rarely. Of course, with extremely bad parameter settings one could generate many random walks in parallel. Here, we will evaluate the parameters experimentally later on.

4.4 Additional Details

We close the discussion of the algorithm by mentioning a few improvements and details that were omitted from Algorithm 2 for the sake of clarity. As mentioned above, each update has a unique id. We use this id in two ways. First, when we restart a walk it carries the id of the (timed-out) update that triggered its

restart. This way, when the walk visits a node where the same update has not yet timed out—recall that we cannot achieve perfect agreement about the age of an update—the update will be forced to time out so the walk is forwarded and not dropped.

Second, each update is accepted only once, that is, in method onReceiveRWProps if the id of the update is the same as that of the current local update rwprops then we drop the received update. This is needed to overcome another problem related to the lack of agreement about update age: it is possible that rwprops has already timed out while rwprops' has not. With our solution it is guaranteed that whenever an update times out, it will not be revived again.

Let us now consider the case where a node rejoins the network after an offline period. In this case, the node waits until it receives a fresh gossip message either via push or pull before taking part in the protocol. This is to prevent premature restarted walks based on outdated updates. Typically, a fresh message will be received almost immediately after joining the network. Note that this fresh message could have the same id as the old update of the joining node, in which case the node will of course participate in the ongoing restarting effort.

In the special case when the joining node was trying to forward a walk when going offline, after receiving the first fresh update it determines whether it is still supposed to forward the same walk, that is, whether the walk is still considered the leader. This situation occurs if the offline period was relatively short, which is a typical situation in, for example, mobile networks.

Finally, we also need to discuss how to start the very first walk. This is a special case because at that point the local variables at the nodes (rw and rwprops) are undefined, so our restarting mechanism is not functional. This can be solved by initializing rw at every node using an initial random walk state and a uniform random negative step count from the interval $[-N, -1]$ where N is the network size, and initializing rwprop to a special update that never times out. Knowing the exact network size is not critical, it can be approximated by piggybacking the push-pull gossip broadcast protocol using known methods [26]. The node that starts the first walk will broadcast an update with step count zero (note that normally only the receiving node creates a new update). We assume that the node that starts the walk is online and connected to the network long enough to broadcast this first update. Of course, when a walk with a negative step count is picked for restarting, it should set the step count to zero.

5 Experimental Analysis of Decentralized Random Walks

We first analyze experimentally the key properties of our random walk protocol without considering machine learning as an application. In Section 6, we complete our experimental analysis by examining our stochastic gradient algorithm. We simulate node churn based on a real trace of smartphone user behavior. To perform the simulations, we employed PeerSim [27].

5.1 Trace Properties

The trace we applied was collected by a locally developed openly available smartphone app called STUNner, as described previously [28]. In a nutshell, the app monitors and collects information about charging status, battery level, bandwidth, and NAT type.

In our study, we had traces of varying lengths taken from 1191 different users. We divided these traces into 2-day segments (with a one-day overlap), resulting in 40,658 segments altogether. With the help of these segments, we were able to simulate a virtual period of up to 2 days by assigning a different segment to each simulated node. When we needed more users than segments, we re-sampled the segments so as to inflate the number of users present artificially.

The churn pattern we obtained is illustrated in Figure 1 based on all the 2-day periods we identified. Although our sample contains users from all over the world, they are mostly from Europe, and some are from the USA. The indicated time is GMT, hence we did not convert times to local time. Also, we treated those users as offline who had a bandwidth of less than 1 Mbit/s.

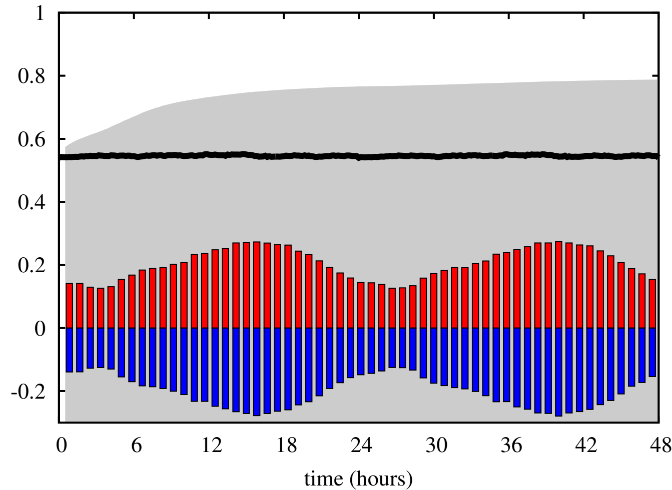

Figure 1: Proportion of users online, and proportion of users that have been online, as a function of time. The bars indicate the proportion of the simulated users that log in and log out (shown as a negative proportion), respectively, in the given period.

Note that we can simulate the case where a participating phone is required to have at least a certain battery level. From the point of view of churn, though, the worst case is when any battery levels are allowed. Thus, we simulate this scenario, noting that conserving the battery is possible and in fact that would also make churn less extreme because it reduces short sessions [28].

5.2 Parameters and Evaluation Metrics

We set $\Delta = 100ms$. Our two free parameters are δ and the random walk transmission time δ_{rw}. Transmission time is a function of the size of the random walk state and the bandwidth allocated to the application; it is more convenient to vary transmission time directly. The network size was $N = 10000$.

In our overlay network every node had 50 fix neighbors. Most of these neighbors were offline at any given time, but the online nodes still formed a connected network. The random walk as well as the gossip messages select a uniform random online neighbor.

The free parameters δ and δ_{rw} took values from $\delta_{rw} \in \{\Delta, 100\Delta\}$ and $\delta \in \{\delta_{rw} + 10\Delta, \delta_{rw} + 20\Delta, \delta_{rw} + 100\Delta\}$ taking all possible 6 combinations. Note that we have to have $\delta \geq \delta_{rw}$ because an update will not be overwritten for a time period of δ_{rw} on average. In addition, δ has to account for the logarithmic dissemination time of the gossip broadcast. Thus, the three values of δ can be considered small, realistic, and large, respectively.

We wish to measure *agility, efficiency* and *longevity*. As a function of time, we recorded the age of the oldest random walk, which characterizes both agility (the steepness of this function) and longevity (the absolute value of this function). We recorded the number of random walks that were propagating concurrently as a function of time, which characterizes efficiency.

5.3 Results

Figure 2 shows our results. The number of random walks is shown as dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of an oldest random walk is shown as colored points, different colors indicating different random walks. We also give the average number of concurrent walks during the experiment in each plot.

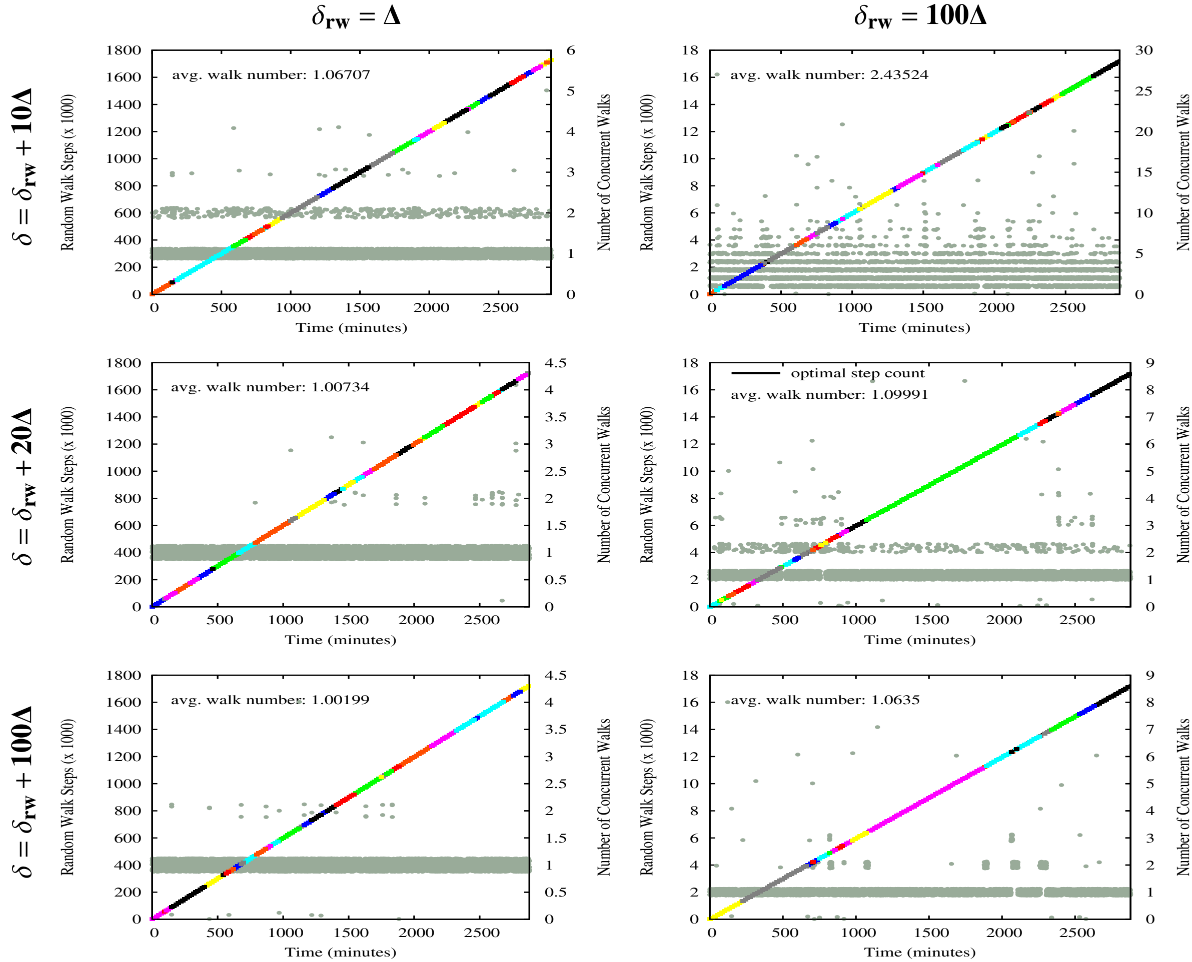

Figure 2: Experiments with all the combinations of δ_{rw} and δ. The number of random walks is shown as green dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of an oldest random walk is shown as colored points, different colors indicate different random walks.

In these simulations agility and longevity are optimal, in the sense that the maximal age grows at the theoretical maximum speed (that speed is in fact indicated by a straight line, which is completely covered by the dots). As expected, the smallest value of δ results in the largest number of restarts, as the gossip broadcast is not always able to converge. Clearly, as we increase δ, the number of random walks quickly decreases to one, sometimes dropping to zero or jumping to two for a very short time. Interestingly, the very large δ does not result in a visible slowdown of the walks. This is because the extinction events are actually quite rare in our simulated trace: they are indicated by the number of walks dropping to zero.

We also wanted to stress-test the algorithm by artificially increasing the number of random walks that die out. For this reason, we artificially killed each random walk with a probability of 5% in each step of the walk, thereby allowing 20 steps on average. This is an extreme and highly unrealistic scenario. We repeated our experiments as shown in Figure 3.

In this extreme scenario the effect of the different parameter settings is more clearly visible. Increasing δ decreases the redundant walks to close to optimal levels, although very rarely for very short periods of time there may be many walks (the plots span the full range covering the outliers as well). On average, however, the efficiency is acceptable. At the same time, the speed of the walks is noticeably reduced.

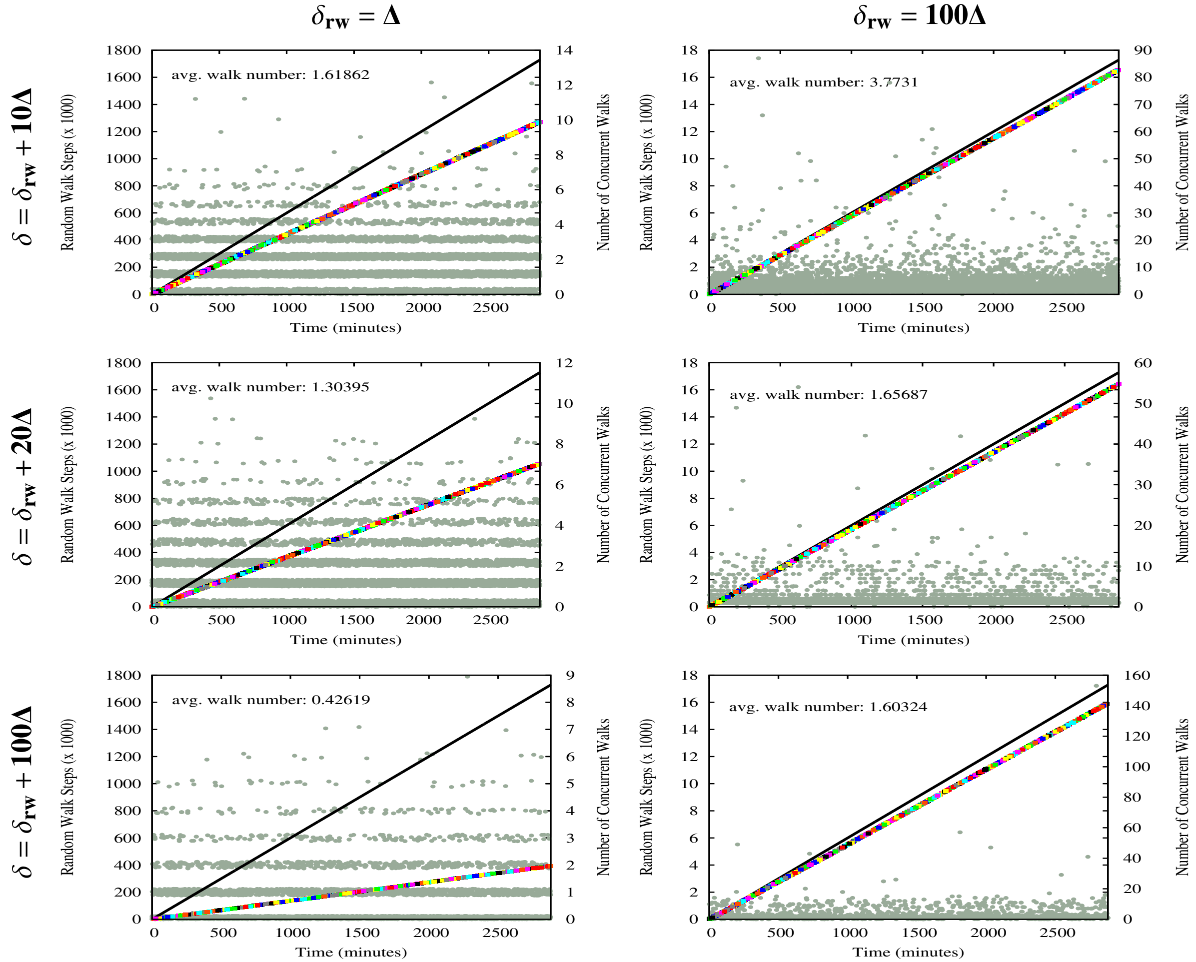

Figure 3: Experiments with 5% drop probability. The number of random walks is shown as green dots (integers, translated slightly vertically by random noise to illustrate density) and the step count of an oldest random walk is shown as colored points, different colors indicate different random walks.

If we increase δ further, we can no longer increase the efficiency, however, the speed of the walks continues to decrease. We stress again that this scenario has been included only to illustrate an extreme corner of our parameter space. Nevertheless, if the random walks go extinct very frequently then the slowing effect of δ becomes more pronounced, and there will be a tradeoff between agility and efficiency. Also, the system is more stable assuming large transmission times (that is, large random walk states), as the average number of walks is close to one.

6 Experimental Analysis of Differentially Private SGD

Here, we present the experimental evaluation of our private gradient methods on realistic, real-life machine learning data sets. The goals of these experiments are twofold. First, we demonstrate the practical feasibility of differential privacy in general under the system assumptions we presented earlier. This includes the speed of convergence and the quality of the end result. Second, we wish to explore several design options and offer practical advice on how to parameterize the differentially private mechanism we propose.

Our experiments will focus on the speed of convergence of the learning algorithm and they will be

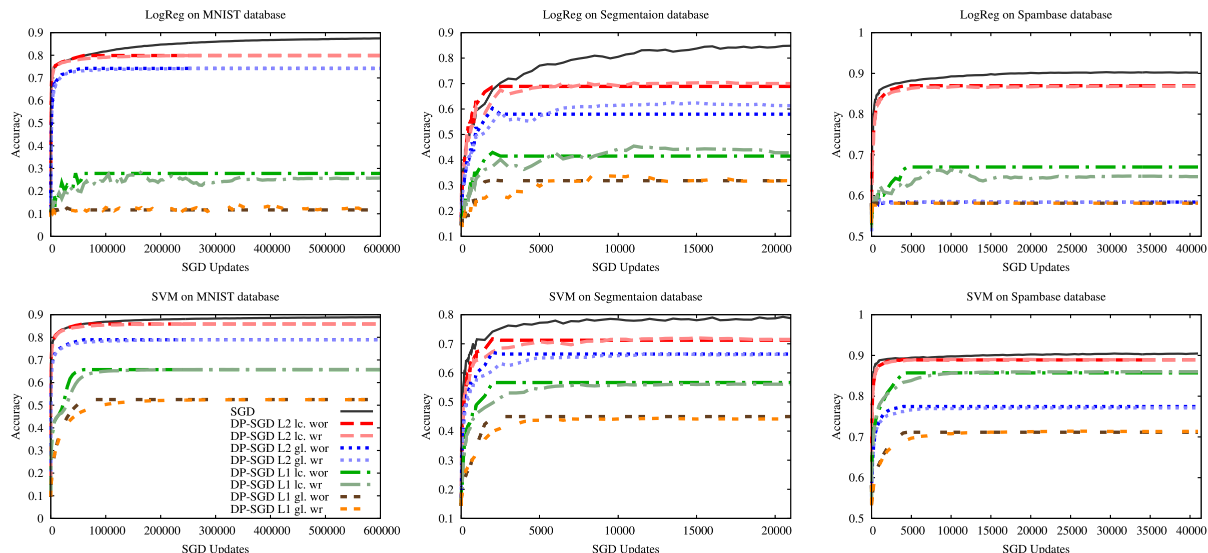

Figure 4: Algorithm DP-SGD-1 with different vector norm (L^1 vs. L^2), normalization (local vs. global), and sampling (with or without replacement) with privacy budget $\epsilon = 1$. The x-axis spans a number of updates ten times the size of the data set.

run with the assumption of an idealized random walk service. In this sense, the experiments described in Sections 5 and 6 are complementary, in the sense that once we have established that we have a system in place that provides a high quality and robust random walk abstraction, it is best to focus on learning performance under an ideal random walk so that we can separate the various factors that affect learning performance. Based on the results presented in Section 5, under realistic parameter settings for δ these results will approximate those over ideal random walks to a very high precision.

6.1 Algorithm Variants

Our algorithm variants are based on the algorithms described earlier in Section 3. First, we will state the parameter settings we used, and then we will introduce the notations for these settings. In every case we run SGD with the differentially private SGD update rule in Equation (8). The learning rate was set to $\eta_t = t^{-\frac{1}{2}}$ and $\lambda = 10^{-4}$. Below, we will refer to this algorithm as DP-SGD.

The privacy budget we used was set to $\epsilon = 1$ unless otherwise stated. We also experimented with $\epsilon = 0.1$ to illustrate a stricter privacy requirement. As for using this budget, we study the three methods presented in Section 3. The first is the method where, for each example, we use up the entire budget ϵ in a single update. Here, we will refer to this variant as DP-SGD-1. The remaining two variants will be referred to as DP-SGD-5 and DP-SGD-∞, respectively. These correspond to the cases when we allow 5 updates using each example, all of which use up a budget of $\epsilon/5$, and when we allow any number of updates with update t using a budget of $\epsilon/2^t$.

The sampling of the next step of the random walk (that is, the next sample to update with) was selected independently at random for each step with or without replacement. In the case of sampling without replacement, when the samples run out, we will restart the sampling with the full set.

As for selecting the vector norm in the definition of global sensitivity, we experiment with the L^2 norm and the L^1 norm. This defines the noise distribution, namely the distribution of N_t in Equation (8), as presented earlier in Section 2.3.

Finally, as a baseline, we also present the performance of SGD without the noise term N_t. We shall

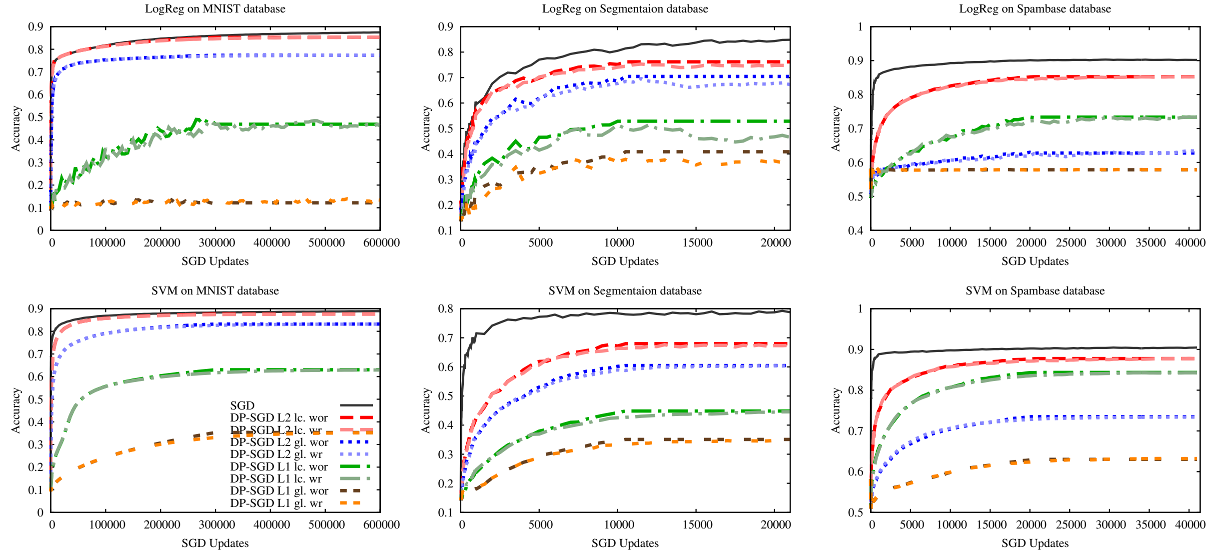

Figure 5: Algorithm DP-SGD-5 with different vector norm (L^1 vs. L^2), normalization (local vs. global), and sampling (with or without replacement) with privacy budget $\epsilon = 1$. The x-axis spans a number of updates ten times the size of the data set.

refer to this variant simply as SGD.

Regarding specific algorithms needed to implement SGD, we include logistic regression and the Pegasos algorithm (see Equations (9) and (10)). Depending on the data set at hand, we employed the multi-class variants of these algorithms. Here, we will refer to these algorithms as LogReg and SVM.

6.2 Data Sets and Preprocessing

We selected our databases from different machine learning domains with different properties. Our first data set, called MNIST [29], contains gray level images of handwritten digits (from 0 to 9) of size 28×28. In the Image Segmentation data set the goal is to assign the pixels to one of the 7 hand-labeled segments, based on a set of high-level features. Finally, in the Spambase data set the task is to detect spam e-mails, again, based on a set of high level features like the frequencies of suspicious words. The last two data sets here form part of the UCI machine learning repository [30]. The main properties of these data sets are summarized in Table 1.

Table 1: The main properties of the data sets

	MNIST	Segmentation	Spambase
Training set size	60 000	2310	4140
Test set size	10 000	210	461
Number of features	784	19	57
Number of classes	10	7	2
Class-label distribution	uniform	uniform	6:4

First, we performed the following preprocessing steps on the data sets. In each case we normalized

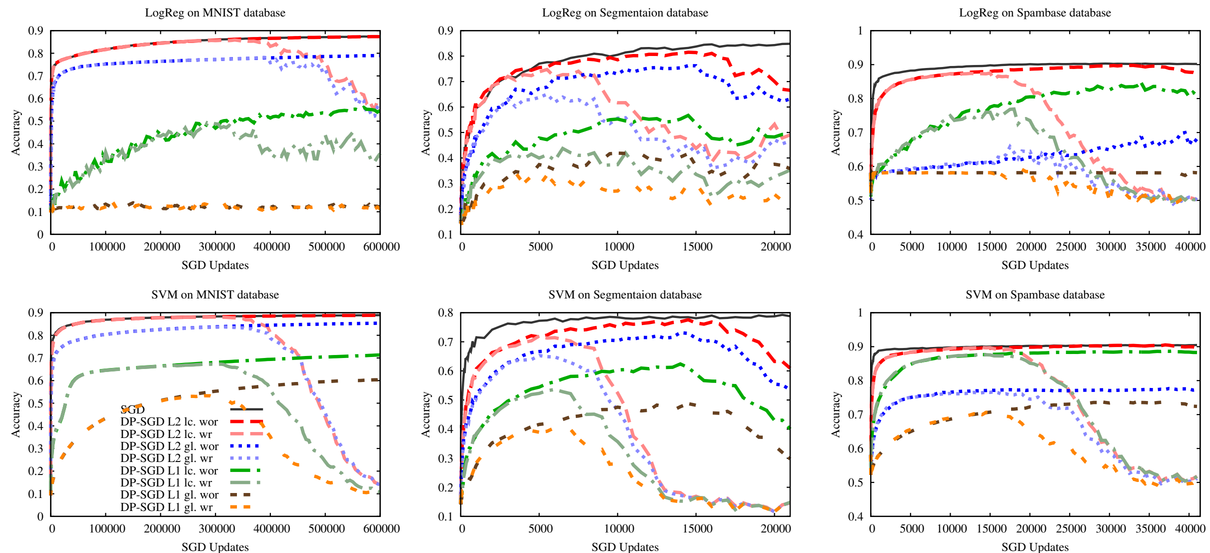

Figure 6: Algorithm DP-SGD-∞ with different vector norm (L^1 vs. L^2), normalization (local vs. global), and sampling (with or without replacement) with privacy budget $\epsilon = 1$. The x-axis spans a number of updates ten times the size of the data set.

the features one-by-one by linearly transforming them into the [0,1] interval, based on the maximum and minimum values of the given feature in the training data set. Then we performed one of the two following normalization steps. As the first option, we projected every feature vector onto the unit ball by normalizing their length to 1. Here, length is understood in terms of the L^2 norm or the L^1 norm, depending on the noise distribution in use. We will call this type of normalization *local normalization*. In this case, the topological structure is somewhat distorted, but the signal-to-noise ratio is maximized as the amount of noise to be added is independent of the individual examples. As the second option, we tested *global normalization*, where the normalization coefficient was globally determined based on the vectors of maximal length. This way, just the lengths of the examples of maximal length become 1, the other vectors becoming shorter than 1 and the topological structure of the data set is preserved. But, in this case, the signal-to-noise ratio might become very low for short vectors. Once again, length is understood in terms of the L^2 norm or the L^1 norm, depending on the noise distribution applied.

6.3 Discussion

In order to measure the performance of the algorithms we computed their classification accuracy, namely the fraction of correctly classified instances:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n}\delta(y_i = f_w(x_i)), \tag{11}$$

where n is the number of test examples. We measured accuracy in regular intervals after a given number of updates. We then plotted the average of 20 independent measurements (where the various measurements differ due to the randomness of sampling the data).

The results of the experiments are shown in three sets of plots in Figures 4, 5 and 6 for the three different algorithms DP-SGD-1, DP-SGD-5 and DP-SGD-∞, respectively. From them, we can draw several interesting conclusions. First, applying the L^2 norm is clearly superior in each set of experiments, independently of how the other parameters are set. The explanation might be that under the L^2 norm, the

added noise is much less likely to be “sparse”; that is, noise is spread more evenly over the coordinates, which in turn causes less disturbance for the gradient steps.

Another clear, and somewhat surprising observation is that it is much better to apply local normalization on the training samples in all the cases we examined. This is good news because, in our distributed setting, local normalization is obviously a local operation and so it is much cheaper to implement than global normalization. The result is somewhat surprising though, because global normalization preserves the topological structure, whereas local normalization does not. Still, the relative noise on a given node might be very large in the case of global normalization since many examples will be shrunk to less than the maximal length. This might a more important factor than the exact topological structure.

The effect of sampling the random walk (i.e., neighbor selection) is more subtle. Looking at algorithms DP-SGD-1 and DP-SGD-5 one might think that there is no significant difference between these sampling methods. However, in the case of DP-SGD-∞ the difference becomes dramatic. After a certain number of iterations sampling with replacement causes a major drop in accuracy. The reason is that with this sampling there will be many nodes that are visited much more often than ten times (note that on average, each node is visited ten times in each run, but with sampling with replacement there is variance in the number of times a node is sampled). At the same time, the privacy budget for step t is only $\epsilon/2^t$, which results in an exponentially increasing amount of noise for each step. Beyond a certain point, the noise becomes so large that the signal-to-noise ratio tends to zero, at which point the accuracy starts to decrease. This is also true for sampling without replacement, only in that case the decrease in performance is seen only later because all nodes are visited an equal number of times so there are no outliers that inject large noise earlier.

The consequence of this last observation is that, although DP-SGD-∞ does indeed achieve the best performance in a number of cases, it is less practical than the other options because one needs to implement sampling without replacement—a non-trivial task in fully distributed networks—and, more importantly, one needs to implement a stopping rule that detects when the accuracy starts to decrease. These goals are not impossible to accomplish, yet the other two algorithms can achieve a similar performance while being much simpler and more robust.

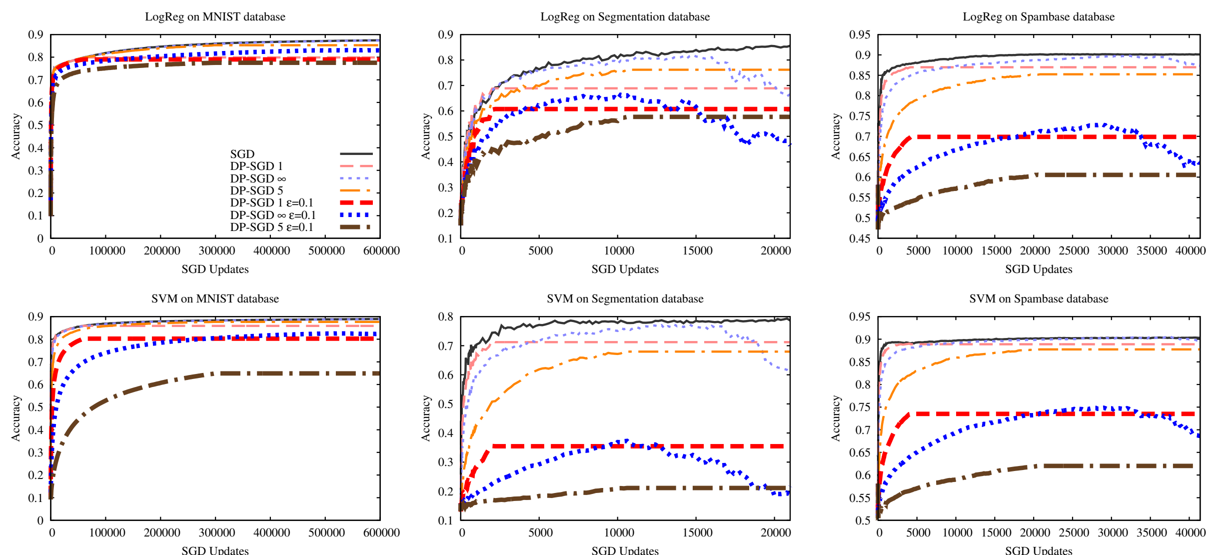

Figure 7: The effect of the lower privacy budget $\epsilon = 0.1$ (i.e., higher privacy requirements) with the L^2 norm, local normalization, and uniform sampling without replacement. Curves with $\epsilon = 1$ are repeated for comparison.

Figure 7 illustrates the effect of the size of the privacy budget on the algorithms. It also allows us to compare the different algorithms directly under the best setting we found empirically in the previous experiments. Manifestly, a lower privacy budget reduces the accuracy of all the algorithms, although to a different degree. Here, DP-SGD-1 seems to be the most robust one. This is because although it performs just a single update step with a given node, that step at least has relatively little noise. Also, the size of the database plays an important role as well. With a small database (small network) we can make fewer gradient updates overall given that we visit each node only once. At the same time, with a very large database one could achieve good convergence due to visiting many samples, even with relatively large noise in each step. This is implied by the theory of SGD and stochastic approximation in general [5].

7 Conclusions

Here, we presented a practical protocol for performing private SGD over an unreliable decentralized system. As part of this, we introduced a protocol to implement a robust random walk that is fast, efficient and long-lived in a dynamic network environment. We simulated the protocol over a smartphone trace and we found that the protocol is robust to its main parameter, the timeout threshold δ, which determines when a random walk is considered dead. We obtained an acceptable performance even in an unrealistic extreme scenario where we artificially removed random walks with a 5% probability in each step. In this case, the protocol is more sensitive to δ, but with a sensible setting a good compromise can be achieved between efficiency and agility.

We then examined the decentralized implementation of private SGD variants. We assumed that there are potentially millions of nodes and each of them contain small amounts of personal data. In the given implementation of SGD, the privacy was realized just based on local processing. Therefore privacy concern only occur if a node is hijacked by a malicious attacker.

Compared with the unprotected distributed SGD algorithm, the examined implementations empirically revealed that the privacy can be maintained with a close-to-optimal accuracy and without communication overheads. To achieve the best performance we proposed to using local normalization and the L^2 norm for the normalization of training data. Furthermore, we recommend that each training data example should be used as little as possible, which is determined by the privacy budget.

It should also be mentioned here that if each node has multiple examples, then batch gradient descent can be applied. In that case, the average batch gradient requires relatively less noise. As a consequence, the models become less noisy and the overall convergence can improve. Having only one record per node is thus the worst case scenario.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. of the 1st Edition of the MCC Workshop on Mobile Cloud Computing (MCC'12), Helsinki, Finland*. ACM, August 2012, pp. 13–16.

[2] A. Rial and G. Danezis, "Privacy-preserving smart metering," in *Proc. of the 10th Annual ACM Workshop on Privacy in the Electronic Society (WPES'11), Chicago, IL, USA*. ACM, October 2011, pp. 49–60.

[3] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, and L. Yang, "Data mining for internet of things: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 77–97, 2014.

[4] A. S. Pentland, "Society's nervous system: Building effective government, energy, and public health systems," *Computer*, vol. 45, no. 1, pp. 31–38, January 2012.

[5] L. Bottou and Y. LeCun, "Large scale online learning," in *Advances in Neural Information Processing Systems 16*, S. Thrun, L. Saul, and B. Schölkopf, Eds. Cambridge, MA: MIT Press, 2004.

[6] R. Ormándi, I. Hegedűs, and M. Jelasity, "Gossip learning with linear models on fully distributed data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 556–571, 2013.

[7] S. Dolev, A. Israeli, and S. Moran, "Uniform dynamic self-stabilizing leader election," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 4, pp. 424–440, April 1997.

[8] C. Dwork, "A firm foundation for private data analysis," *Communications of the ACM*, vol. 54, no. 1, pp. 86–95, January 2011.

[9] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "Gupt: privacy preserving data analysis made easy," in *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12), Helsinki, Finland*. ACM, August 2012, pp. 349–360.

[10] A. Friedman and A. Schuster, "Data mining with differential privacy," in *Proc. of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'10), Washington, DC, USA*. ACM, July 2010, pp. 493–502.

[11] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate, "Differentially private empirical risk minimization," *Journal of Machine Learning Research*, vol. 12, pp. 1069–1109, July 2011.

[12] K. Chaudhuri and C. Monteleoni, "Privacy-preserving logistic regression," in *Advances in Neural Information Processing Systems 21, (NIPS)*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds. Curran Associates, Inc., 2009, pp. 289–296. [Online]. Available: http://papers.nips.cc/paper/3486-privacy-preserving-logistic-regression.pdf

[13] S. Song, K. Chaudhuri, and A. D. Sarwate, "Stochastic gradient descent with differentially private updates," in *Proc. of the 2013 IEEE Global Conference on Signal and Information Processing (GlobalSIP'13), Austin, TX, USA*, December 2013, pp. 245–248.

[14] I. Hegedűs and M. Jelasity, "Distributed differentially private stochastic gradient descent: An empirical study," in *Proc. of the 24th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'16), Heraklion, Greece*. Heraklion, Greece: IEEE, February 2016, pp. 566–573.

[15] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade, Second Edition*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer Berlin Heidelberg, 2012, vol. 7700, pp. 421–436.

[16] T. M. Mitchell, *Machine Learning*, 2nd ed., E. M. Munson, Ed. New York: McGraw-Hill, 1997.

[17] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: primal estimated sub-gradient solver for SVM," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2010.

[18] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer-Verlag New York, 2006.

[19] K. Crammer and Y. Singer, "On the algorithmic implementation of multiclass kernel-based vector machines," *Journal of Machine Learning Research*, vol. 2, pp. 265–292, March 2002.

[20] N. Tölgyesi and M. Jelasity, "Adaptive peer sampling with newscast," in *Proc. of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par'09), Delft, The Netherlands*, ser. Lecture Notes in Computer Science, vol. 5704. Springer Berlin Heidelberg, August 2009, pp. 523–534.

[21] K. Birman, M. Jelasity, R. Kleinberg, and E. Tremel, "Building a secure and privacy-preserving smart grid," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 131–136, January 2015.

[22] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proc. of the 3rd Theory of Cryptography Conference (TCC'06), New York, NY, USA*, ser. Lecture Notes in Computer Science, S. Halevi and T. Rabin, Eds., vol. 3876. Springer Berlin Heidelberg, March 2006, pp. 265–284.

[23] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in *Proc. of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09), Providence, USA*. ACM, June-July 2009, pp. 19–30.

[24] D. Stutzbach, R. Rejaie, N. Duffield, S. Sen, and W. Willinger, "On unbiased sampling for unstructured peer-to-peer networks," *IEEE/ACM Transactions on Networking*, vol. 17, no. 2, pp. 377–390, April 2009.

[25] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking, "Randomized rumor spreading," in *Proc. of the 41st Annual Symposium on Foundations of Computer Science (FOCS'00), Redondo Beach, CA, USA*. Washington, DC, USA: IEEE, November 2000, pp. 565–574.

[26] E. Le Merrer, A.-M. Kermarrec, and L. Massoulie, "Peer to peer size estimation in large and dynamic networks: A comparative study," in *Proc. of the 15th 15th IEEE International Conference on High Performance Distributed Computing (HPDC'06), Paris, France*. IEEE, June 2006, pp. 7–17.

[27] A. Montresor and M. Jelasity, "Peersim: A scalable P2P simulator," in *Proc. of the 9th IEEE International*

Conference on Peer-to-Peer Computing (P2P'09), Seattle, WA, USA. IEEE, September 2009, pp. 99–100, extended abstract.

[28] Á. Berta, V. Bilicki, and M. Jelasity, "Defining and understanding smartphone churn over the internet: a measurement study," in *Proc. of 14th IEEE International Conference on Peer-to-Peer Computing (P2P'14), London, UK*. IEEE, September 2014, pp. 1–5.

[29] Y. Lecun, C. Cortes, and B. Christopher, J.C., "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/ [Online; Accessed on June 10, 2016].

[30] K. Bache and M. Lichman, "UCI machine learning repository, university of california, irvine, school of information and computer sciences," 2013, http://archive.ics.uci.edu/ml [Online; Accessed on June 10, 2016].

Author Biography

István Hegedűs is a PhD student at the University of Szeged under the supervision of M. Jelasity and works as a research assistant in the MTA-SZTE Research Group on Artificial Intelligence at the University of Szeged. His main research interests are fully distributed algorithms and machine learning.

Árpád Berta is a PhD student at the University of Szeged under the supervision of M. Jelasity. His main research interests are collaborative mobile gossip learning and fully distributed data mining.

Márk Jelasity is a senior researcher in the MTA-SZTE Research Group on Artificial Intelligence at the University of Szeged. He obtained his PhD degree from the University of Leiden in 2001, and his DSc degree from the Hungarian Academy of Sciences in 2015. His research interests include decentralized algorithms for data aggregation and mining in large scale unreliable systems, data privacy, and self-organizing systems.