Efficient Gossip Algorithms for Machine Learning

PhD Thesis

Gábor Danner
Supervisor: Márk Jelasity, PhD

Doctoral School of Computer Science
Department of Computer Algorithms and Artificial Intelligence
Faculty of Science and Informatics
University of Szeged

Szeged
2022

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Márk Jelasity, for guiding and supporting my research and providing the opportunity to work on fascinating topics.

Next, I would like to thank my colleagues who helped me to discover interesting areas of science and helped give birth to new ideas during our discussions. In alphabetical order, they are Dr. Árpád Berta, Dr. István Hegedűs and Dr. Róbert Ormándi. I would also like to thank David P. Curley for correcting this thesis from a linguistic point of view.

Last but not least, I would like to thank my family and friends for supporting me all this time.

This research work was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 ("Internet of Living Things"), by the Hungarian Ministry of Human Capacities (grant 20391-3/2018/FEKUSTRAT), by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary, and by the University of Szeged Open Access Fund (Grant Number 4411).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent times, smart devices have become part of our daily lives. Their widespread presence offers numerous applications that make use of data mining. Usually, the data is collected at a central location for processing. However, this is becoming increasingly problematic due to the growing privacy concerns of the public and the policy-makers. This has motivated the need for collaborative methods that do not need the collection of sensitive data.

Google proposed federated learning to address this problem. Although this is still a centralized approach, the data remains on the device. It works in a similar way to the parameter server architecture. The server periodically sends the current model to the nodes. They perform an update step using the local data and upload it to the server for aggregation. Compression techniques can be employed to reduce communication costs.

In contrast, we proposed gossip learning, which is fully distributed. The nodes communicate directly, exchanging their models. The lack of need for a central server makes this an attractive approach for startups and communities with limited resources. It can assist the creation of non-profit intelligent smartphone services. It is suitable for other platforms as well, like smart metering and the Internet of Things. Its performance is comparable to that of federated learning in several scenarios.

To improve upon existing gossip learning methods, we introduce a number of techniques that also have applications beyond this. Our research touches upon many topics including secure multiparty computation, communication flow control in decentralized systems, and efficient average consensus algorithms using stateful encoder-decoder pairs.

The goal of secure multiparty computation (MPC) is to compute a function of the private inputs of the parties in such a way that, at the end of the computation, no party knows anything except what can be determined from the result and its own input. Secure sum computation is an important application of secure MPC. However, in our dynamic and unreliable application domain, known MPC algorithms are not scalable or not robust enough. We introduce a quick method to securely compute the sum over a subset of participants, and use it in the calculation of mini-batches in decentralized learning.

Many decentralized algorithms allow both proactive and reactive implementations regarding flow control. Examples include gossip protocols for broadcasting and decentralized computing, as well as chaotic matrix iteration algorithms. In proactive systems, nodes communicate at a fixed rate in regular intervals, while in reactive systems they communicate in response to certain events such as the arrival of fresh data. Although reactive algorithms tend to stabilize/converge/self-heal much faster, they have serious drawbacks: they may cause uncontrolled bursts in bandwidth consumption, and they may also cause starvation when the number of messages circulating in the system becomes too low. Proactive algorithms do not have these problems, but nodes waste a lot of time sitting on fresh information. We propose an adaptive method that combines the advantages of the two approaches.

Mean estimation, also known as average consensus, is an important computational primitive in decentralized systems, for example, in the context of machine learning. When the average of large vectors has to be computed, as in distributed data mining applications, reducing the communication cost becomes a key design goal. One way of reducing communication cost is to add dynamic stateful encoders and decoders to traditional mean estimation protocols. In this approach, each element of a vector message is encoded in a few bits (often only one bit) and decoded by the recipient node. However, due to this encoding and decoding mechanism, these protocols are much more sensitive to benign failure such as message drop and message delay. Properties such as mass conservation are harder to guarantee. Hence, known approaches are formulated under strong assumptions such as reliable communication, atomic non-overlapping transactions or even full synchrony. We propose an efficient algorithm that does not need such assumptions.

This thesis is structured as follows. In Chapter 2, we give an overview of the background.

In Chapter 3, we focus on privacy and security issues. We propose a light-weight protocol to quickly and securely compute the sum query over a subset of participants assuming a semi-honest adversary. During the computation the participants learn no individual values. We apply this protocol to efficiently calculate the sum of gradients as part of a fully distributed minibatch stochastic gradient descent algorithm. The protocol achieves scalability and robustness by exploiting the fact that in this application domain a “quick and dirty” sum computation is acceptable. We utilize the Paillier homomorphic cryptosystem as part of our solution combined with extreme lossy gradient compression to make the cost of the cryptographic algorithms affordable. We demonstrate both theoretically and experimentally that the protocol is indeed practically viable.

Table 1.1: *Correspondence between the chapters and the publications of the author. ● and ○ indicate the core and the related publications, respectively.*

	Chapter 3	Chapter 4	Chapter 5	Chapter 6
DAIS 2015 [21]	●			
SCN 2018 [19]	●			
DAIS 2019 [40]		●		
JPDC 2021 [41]		●	●	
ECML 2019 [39]		○		
ICDCS 2018 [23]			●	
Euro-Par 2018 [22]				●
DICG 2020 [20]				●

In Chapter 4, we present a systematic comparison of gossip learning and federated learning. We examine the aggregated cost for several algorithm-variants in various simulation scenarios. These experimental scenarios include different network sizes and different distributions of the training data over the devices.

In Chapter 5, we propose a family of adaptive flow control protocols that apply rate limiting inspired by the token bucket algorithm, but they also include proactive communication to prevent starvation. With the help of our traffic shaping service, some applications approach the speed of the reactive implementation, while maintaining strong guarantees regarding the total communication cost and burstiness. We perform simulation experiments in different scenarios.

In Chapter 6, we propose a communication efficient algorithm for mean estimation that supports known codecs even when transactions overlap and the nodes are not synchronized. The algorithm is based on push-pull averaging, with novel features to support fault tolerance and compression. As an independent contribution, we also propose a novel codec, called the pivot codec. We demonstrate experimentally that our algorithm improves the performance of existing codecs and the novel pivot codec dominates the competing codecs in the scenarios we studied. We also examine its application in decentralized machine learning. In addition, we also rely on transfer learning for extra compression. This means that we train a relatively small model on top of a high quality pre-trained feature set that is fixed. We demonstrate these contributions via an experimental analysis.

Lastly, in chapters 7 and 8, we give summaries of our contributions in English and Hungarian, respectively.

Chapter 2

Background

In this chapter, we present an introduction to concepts and techniques that are necessary to understand our contributions. First, we give a concise summary of the relevant machine learning concepts. Then, we describe the system model we use. After that, we give a short overview of gossip learning. Finally, we discuss the smart phone trace we used to model node availability.

2.1 Machine Learning Basics

We are concerned with the classification problem, where we are given a data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n examples. An example (x, y) consists 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 set of class labels.

The problem is to find the parameters w of a function $f_w : R^d \to C$ that can correctly classify as many examples as possible in D, as well as outside D, drawn from the same distribution (this latter property is called generalization). Expressed formally, we wish to minimize an objective function $J(w)$ in w:

$$w^* = \arg\min_w J(w) = \arg\min_w \frac{1}{n} \sum_{i=1}^{n} \ell(f_w(x_i), y_i) + \frac{\lambda}{2}\|w\|^2, \tag{2.1}$$

where $\ell()$ is the loss function (the error of the prediction), $\|w\|^2$ is the regularization term, and λ is the regularization coefficient.

Stochastic gradient descent (SGD) [12] is a popular approach for finding w^*. Here, we start with some initial weight vector w_0, and we apply the following update rule repeatedly:

$$w_{t+1} = w_t - \eta_t(\lambda w_t + \frac{\partial \ell(f_w(x_i), y_i)}{\partial w}(w_t)). \tag{2.2}$$

Here, η_t is called the learning rate. This update rule requires a single example (x_i, y_i), and for each update we can choose a random example. A popular way to accelerate the convergence is the use of mini-batches, that is, to update the model with the gradient of the sum of the loss functions of a few training examples (instead of only one) in each iteration. This allows for fast distributed implementations as well [36].

In this thesis we primarily use *logistic regression* as our machine learning model, where the specific form of the objective function is given by

$$J(w,b) = \frac{\lambda}{2}\|w\|^2 - \frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln f_{(w,b)}(x_i) + (1-y_i)\ln(1-f_{(w,b)}(x_i))\right], \tag{2.3}$$

where $y_i \in \{0,1\}$ and

$$f_{(w,b)}(x_i) = P(y_i = 1|x_i, w, b) = \frac{1}{1+e^{(w^T x_i + b)}}. \tag{2.4}$$

Note that $P(y_i = 0|x_i, w, b) = 1 - P(y_i = 1|x_i, w, b)$. In fact, the loss function above is the log-likelihood of the data under this probabilistic model. The parameter b is called the bias of the model. This is often incorporated into w, in which case x is extended with a corresponding component that is always 1. This way, the update rule can be written as

$$w_{t+1} = w_t - \eta_t(\lambda w_t - (f_{w_t}(x_i) - y_i)x_i). \tag{2.5}$$

While our methods can be applied to solve any model-fitting optimization problem, we evaluated them only with linear models. This might seem restrictive, however, the practical applicability of such simple models are greatly extended if one uses them in the context of transfer learning [76]. The idea is that arbitrarily complex pre-trained machine learning models are used as feature extractors, over which a simple (often linear) model is trained over a given new dataset. In linguistic applications, this is becoming a very popular approach, often using BERT [27] as the pre-trained model. This can approximate the performance of training the entire complex model over the new dataset, while using far fewer resources.

2.2 System Model

We model our system as a large set of nodes that communicate via message passing. The protocols we discuss send very large messages, so the delay of successfully delivered messages is determined by the message size and the available network bandwidth (as opposed to network latency). Unless stated otherwise, we evaluate our protocols by assuming a reliable transfer protocol, which implies that we do not consider message drop failure. However, the protocols themselves do not require this

Algorithm 2.1 Gossip Learning Framework

```
 1: $(x, y) \leftarrow$ local training example
 2: currentModel $\leftarrow$ init()
 3: **loop**
 4:     wait($\Delta$)
 5:     $p \leftarrow$ selectPeer()
 6:     send currentModel to $p$
 7: **end loop**
 8: **procedure** ONMODEL($m$)
 9:     $m \leftarrow$ merge($m$, currentModel)
10:     $m \leftarrow$ update($m, x, y$)
11:     currentModel $\leftarrow m$
12: **end procedure**
```

assumption. Nodes are allowed to leave the network at any time and node failures are also permitted. We do not require time to be synchronized over the nodes but we do assume the existence of a local clock.

At every point in time each node is assumed to have a set of neighbors, typically a small subset of the nodes. We assume that the failure of a neighbor is detected by the node. The neighbor set can change over time, but nodes can send messages only to their current neighbors. The set of neighbors might be a uniform random sample from the network or it might be defined by any fixed overlay network, depending on the application. When sending a message, nodes access their neighbors via a peer sampling service. In this thesis we treat this service as a black box, noting that many implementations are available [45, 71] that might depend on the given networking environment and the application requirements as well.

2.3 Gossip Learning

Gossip Learning is a machine learning approach over fully distributed data without central control [65]. Here, we present the simplest variant. We assume that the data set is horizontally distributed over a set of nodes, with each node storing a training example (x, y). The task is to collectively find a machine learning model in such a way that emulates the case when the data set is stored centrally.

The basic idea is that in the network many models perform random walks and are updated at every node using the local example. More precisely, every node executes Algorithm 2.1. First, the node initializes its local model. The model is then periodically sent to another node in the network. When a node receives a model, it merges it into its own, updates it by its locally stored training example using the SGD update rule, and then stores the updated model as its new local model. The most

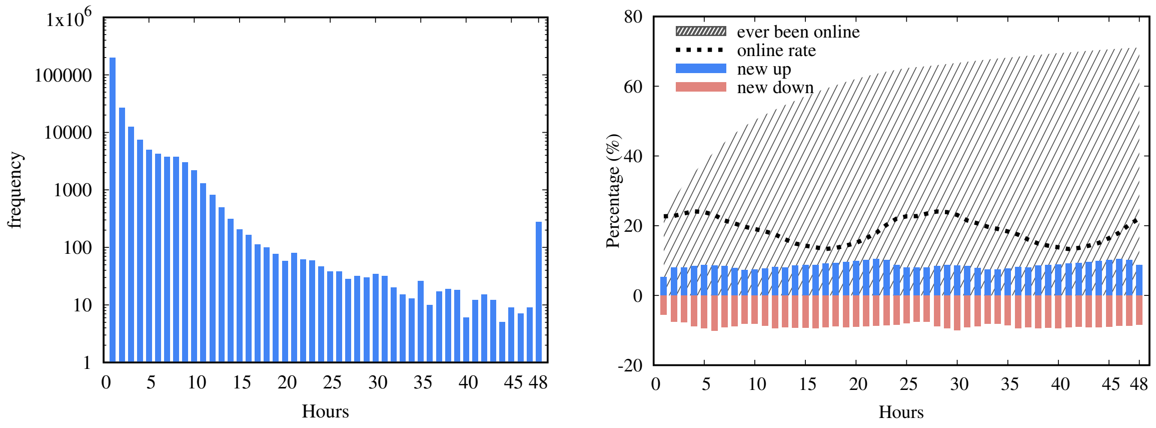

Figure 2.1: *Online session length histogram (left) and device churn (right).*

trivial implementation of MERGE is to simply pick the newly received model, but it is usually better to take the average of the model parameters. Note that the rounds are not synchronized, although all the nodes use the same period Δ.

2.4 Smartphone Trace

We used a trace collected by STUNner, a locally developed, openly available smartphone application [6]. In short, the app monitors and collects information about the battery level, charging status, bandwidth, and NAT type.

The trace contains time series spanning varying lengths of time, originating from 1191 different users. Based on the UTC hour of day, we split the data into 2-day segments (with a one-day overlap), resulting in 40,658 segments altogether. Using this, we can simulate a virtual 48-hour period by assigning a randomly selected segment to each simulated node. The sampling of the segments is done without replacement. When the pool of segments runs out (which happens when we need more nodes than there are segments) we re-initialize the pool with the original set of segments and continue the sampling without replacement.

To make our algorithms phone and user friendly, we consider a device to be online (available) when it has been on a charger and connected to the Internet (with a bandwidth of at least 1 Mbit/s) for at least a minute, therefore we do not use battery power at all.

The main properties of the trace are shown in Figure 2.1. The plot on the right illustrates churn by showing what percentage of the nodes left, or joined the network (at least once) in any given hour. Notice that at any given moment about 20% of the nodes are online. The mean online session length is 81.37 minutes.

Chapter 3

Gossip Learning with Privacy Preservation

Data mining over personal data harvested from mobile devices is a very sensitive problem due to the strong requirements of privacy preservation and security. Recently, the *federated learning* approach was proposed to solve this problem by not collecting the data in the first place but instead processing the data in place and creating the final models in the cloud based on the models created locally [50, 58].

We go one step further and propose a solution that does not utilize centralized resources at all. The main motivation for a fully distributed solution in our cloud-based era is to preserve privacy by avoiding the central collection of any personal data, even in pre-processed form. Another advantage of distributed processing is that this way we can make full use of all the local personal data, which is impossible in cloud-based or private centralized data silos that store only specific subsets of the data. The key issue here of course is to offer decentralized algorithms that are competitive with approaches like federated learning in terms of time and communication complexity, and that provide increased levels of privacy and security.

Previously, we proposed numerous distributed machine learning algorithms in a framework called gossip learning. In this framework models perform random walks over the network and are trained using stochastic gradient descent [65] (see Section 2.3). This involves an update step in which nodes use their local data to improve each model they receive, and then forward the updated model along the next step of the random walk. Assuming the random walk is secure—which in itself is a research problem on its own, see e.g. [46]—it is hard for an adversary to obtain the two versions of the model right before and right after the local update step at any given node. This provides reasonable protection against uncovering private data.

However, this method is susceptible to collusion. If the nodes before and after an update in the random walk collude they can recover private data. In this chapter we address this problem, and improve gossip learning so that it can tolerate a much higher proportion of honest but curious (or semi-honest) adversaries. The key idea behind the approach is that in each step of the random walk we form a group of peers that securely compute the sum of their gradients, and the model update step is performed using this aggregated gradient. In machine learning this is called mini-batch learning, which—apart from increasing the resistance to collusion—is known to often speed up the learning algorithm as well (see, for example, [25]).

It might seem attractive to run a secure multiparty computation (MPC) algorithm within the mini-batch to compute the sum of the gradients. The goal of MPC is to compute a function of the private inputs of the parties in such a way that at the end of the computation, no party knows anything except what can be determined from the result and its own input [88]. Secure sum computation is an important application of secure MPC [18].

However, we do not only require our algorithm to be secure but also fast, light-weight, and robust, since the participating nodes may go offline at any time [6] and they might have limited resources. One key observation is that for the mini-batch algorithm we do not need a precise sum; in fact, the sum over any group that is large enough to protect privacy will do. At the same time, it is unlikely that all the nodes will stay online until the end of the computation. We propose a protocol that—using a binomial tree topology and Paillier homomorphic encryption—can produce a "quick and dirty" partial sum even in the event of failures, has adjustable capability of resisting collusion, and can be completed in logarithmic time.

We also put a great emphasis on demonstrating that the proposed protocol is practically viable. This is a non-trivial question because homomorphic cryptosystems can quickly become very expensive when applied along with large-enough key-sizes (such as 2048 bit keys), especially considering that in machine learning the gradients can be rather large. To achieve practical viability, we propose an extreme lossy compression, where we discretize floating point gradient values to as few as two bits. We demonstrate experimentally that this does not affect learning accuracy yet allows for an affordable cryptography cost. Our simulations are based on a real smartphone trace we collected [6], described in Section 2.4.

3.1 Related Work

There are many approaches that have goals similar to ours, that is, to perform computations over a large and highly distributed database or network in a secure and privacy preserving way. Our work touches upon several fields of research including machine learning, distributed systems and algorithms, secure multiparty computation and privacy. Our contribution lies in the intersection of these areas. Here we focus only on related work that is directly relevant to our present contributions.

Algorithms exist for completely generic secure computations, Saia and Zamani give a comprehensive overview with a focus on scalability [72]. However, due to their focus on generic computations, these approaches are relatively complex and in the context of our application they still do not scale well enough, and do not tolerate dynamic membership either.

Approaches targeted at specific problems are more promising. Clifton et al. propose, among other things, an algorithm to compute a sum [18]. This algorithm requires linear time in the network size and it does not tolerate node failure either. Bickson et al. focus on a class of computations over graphs, where the computation is performed in an iterative manner through a series of local updates [7]. They introduce a secure algorithm to compute local sums over neighboring nodes based on secret sharing. Unfortunately, this model of computation does not cover our problem as we want to compute mini-batches of a size independent of the size of the direct neighborhood, and the proposed approach does not scale well in that sense. Besides, the robustness of the method is not satisfactory either [61]. Han et al. address stochastic gradient search explicitly [37]. However, they assume that the parties involved have large portions of the database, so their solution is not applicable in our scenario.

Bonawitz et al. [11] address a similar problem setting where the goal is to compute a secure sum in an efficient and robust manner. They also assume a semi-honest adversarial model (with a limited set of potentially malicious behaviors by a server). However, their solution requires a server and an all-to-all broadcast primitive even in the most efficient version of their protocol. Our solution requires a linear number of messages only.

The algorithm of Ahmad and Khokhar is similar to ours [1], as they also use a tree to aggregate values using homomorphic encryption. However, in their solution all the nodes have the same public key and the private key is distributed over a subset of elite nodes using secret sharing. The problem with this approach in our mini-batch gradient descent application is that for each mini-batch a new key set has to be generated for the group, which requires frequent access to a trusted server, otherwise the method is highly vulnerable in the key generation phase. In our solution, all the nodes have their own public/private key pair and no keys have to be shared at any point in time. Besides, these key pairs may remain the same in every mini-batch the given node participates in without compromising our security guarantees.

We need to mention the area of differential privacy [30], which is concerned with the the problem that the (perhaps securely computed) output itself might contain information about individual records. The approach is that a carefully designed noise term is added to the output. Gradient search has been addressed in this framework (for example, [68]). In our distributed setup, this noise term can be computed in a distributed and secure way [31].

3.2 Adversarial Model

We assume that the adversaries are honest but curious (or semi-honest). That is, nodes corrupted by an adversary will follow the protocol but the adversary can see the internal state of the node. The goal of the adversary is to learn about the private data of other nodes (note that the adversary can obviously see the private data on the node it observes directly). Wiretapping is allowed, since all the sensitive messages in our protocol are encrypted.

We assume a static adversarial model, which means that the corrupted nodes are picked a priori, independently of the state of the protocol or the network. As of the number of corrupted nodes, we will consider the threshold model, in which at most a given number of nodes are corrupted, as well as a probabilistic model, in which any node can be corrupted with a given constant probability [57].

We also assume that adversaries are not able to manipulate the set of neighbors. In each application domain this assumption translates to different requirements. For example, if an overlay service is used to maintain the neighbors then this service has to be secure itself.

3.3 Our Solution

Our approach here is based on the GOLF framework, outlined in Section 2.3, replacing the local update step with a mini-batch approach: at each step, when a node receives a model to update, it coordinates the distributed computation of a mini-batch gradient and then uses this gradient to update the model. Based on the assumptions in Sections 2.2 and 3.2 we now present our algorithm for computing a mini-batch gradient.

3.3.1 Mini-batch Tree Topology

The very first step for computing a mini-batch gradient is to create a temporary group of random nodes that form the mini-batch. In our decentralized environment we do this by building a rooted overlay tree. The basic version of our algorithm will require the overlay tree not only to be rooted at the node computing the gradient but also to be *trunked*.

Definition 1 (trunked tree). *Any rooted tree is* 1-trunked. *For $k > 1$, a rooted tree is k-*trunked *if the root has exactly one child node, and the corresponding subtree is a $(k-1)$-trunked tree.*

Let N denote the intended size of the mini-batch group. We assume that N is significantly less than the network size. Let S be a parameter that determines the desired security level ($N \geq S \geq 2$). We can now state that we require an *S-trunked tree* rooted at the node that is being visited by gossip learning. As we will see later, this is to prevent a malicious root to collect too much information.

Apart from the trunk, the tree can be arbitrary, however, we propose a *binomial tree* as a preferable choice. If every node already in the tree spawns a new child node in periodic rounds (starting from a single root node) then the result is a binomial tree. It is not possible to construct a tree of a given size faster, since in the case of a binomial tree each node keeps working continuously so the efficiency is maximal. Of course we assumed here that child nodes can be added only sequentially at a given node. However, if we also assume that all the nodes have the same up- and download bandwidth cap then adding nodes in parallel will be proportionally slower thus parallelism provides no advantage as long as we utilize the maximal available bandwidth. The same up- and download bandwidth requirement is naturally satisfied in our application domain because we assume that the protocol is allowed to use only a fixed, relatively small amount of bandwidth (such as 1 Mbps) and low bandwidth connections are excluded from the set of possible overlay connections.

Another advantage of binomial trees is that we can use the links in reverse order of construction for uploading and aggregating data along the tree. This way, we get a data aggregation schedule that is similarly efficient and also collision-free in the sense that each node communicates with at most one node at a given time.

The tree overlay network we have described so far can be constructed over a random overlay network by first building the trunk (which takes a random walk of $S-1$ steps) and then recursively constructing a binomial tree of depth D, resulting in an S-trunked tree of size $2^D + S - 1$ and total depth $d = D + S - 1$. Every child node is chosen randomly from those neighbors of the node that are both online and not in the tree already. No attention needs to be paid to reliability. We generate the tree quickly and use it only once quickly. Normally, some subtrees will be lost in the process because of churn but our algorithm is designed to tolerate this. The effect of certain parameters, such as the binomial tree parameter and node failures, will be discussed later in the evaluation.

3.3.2 Calculating the Gradient

The sum we want to calculate is over vectors of real numbers. Without loss of generality, we discuss the one-dimensional case from now on for simplicity. Homomorphic encryption works over integers, to be precise, over the set of residue classes $\mathbb{Z}_n$ for some large n. For this reason we need to discretize the real interval that includes all possible sums we might calculate, and we need to map the resulting discrete intervals to residue classes in $\mathbb{Z}_M$ where M defines the granularity of the resolution of the discretization. This mapping is natural, we do not go into details here. Since the gradient of the loss function for most learning algorithms is bounded, this is not a practical limitation. Also, in Section 3.5 we evaluate the effect of discretization on learning performance and we show that even an extreme compression (discretizing the gradient down to two bits) is tolerable due to the high robustness of the mini-batch gradient method itself.

In a nutshell, the basic idea of the algorithm is to divide the local value at each node into S shares, encrypt these with asymmetric additively homomorphic encryption (such as the Paillier cryptosystem), and send them to the root via the chain of ancestors. Although the shares travel together, they are encrypted with the public keys of different ancestors. Along the route, the arrays of shares are aggregated, and periodically re-encrypted. Finally, the root calculates the sum.

The algorithm consists of three procedures, shown in Algorithm 3.1. These are run locally on the individual nodes. Procedure INIT is called once after the node becomes part of the tree. Here, the function call ANCESTOR(i) returns the descriptor of the ith ancestor on the path towards the root. The descriptor contains the necessary public keys as well. During tree building this information can be given to each node so the nodes can look up the keys of their ancestors locally. For the purposes of the ANCESTOR function, the parent of the root is defined to be itself. Function ENCRYPT(x, y) encrypts the integer x with the public key of node y using an asymmetric additively homomorphic cryptosystem.

Procedure ONMESSAGERECEIVED is called whenever a message is received by the node. A message contains an array of dimension S that contains shares encoded for the S closest ancestors to the sender child. The first element (msg[1]) is thus encrypted for the current node, so it can decrypt it. The rest of the shares are shifted down by one position and added (with homomorphic encryption) to the local array of shares to be sent (operation $a \oplus b$ performs the homomorphic addition of the two encrypted integers a and b to get the encrypted form of the sum of these integers). Note that the ith element ($1 \leq i \leq S-1$) of the array SHARES is encrypted with the public key of the ith ancestor of the current node and is used to aggregate a share of the sum of the subtree except the local value of the current node. The Sth share is aggregated in variable KNOWNSHARE unencrypted. The value of share[S] is not modified in this method, it will be initialized using KNOWNSHARE after all the child nodes that are alive have responded.

After all the shares have been processed, procedure ONNOMOREMESSAGESEXPECTED is called. This happens when the node has received a message from all of its children, or when the remaining children are considered to be dead by a failure detector. The timeout used here has to take into account the depth of the given subtree and the maximal delay of a message. In the case of leaf nodes, this procedure is called right after INIT. When calling ONNOMOREMESSAGESEXPECTED, we know that the ith element ($1 \leq i \leq S-1$) of the array SHARES already contains the ith share of the sum of the subtree rooted at the current node (except the local value of the current node) encrypted with the public key of the ith ancestor of the current node. We also know that KNOWNSHARE contains the Sth share of the same sum unencrypted.

Now, if the current node is the root then the elements of the received array are decrypted and summed. The root can decrypt all the elements because it is the parent of itself, so all the elements are encrypted for the root when the message reaches it. Here, DECRYPT(x) decrypts x using the private key of the current node. Function PUBLISH(x) announces x, the output of the algorithm, that is, the final unencrypted sum.

If the current node is not the root then the local value has to be added, and the Sth element of the array has to be filled. First, the local value is split into S shares according to the S-out-of-S secret-sharing scheme discussed in [57]: $S-1$ out of the S shares are uniformly distributed random integers between 0 and $M-1$. The last share is the difference between the local value and the sum of the random numbers (mod M). This way, the sum of shares equals the local value (mod M). Also, the sum of any non-empty proper subset of these shares is uniformly distributed, therefore nothing can be learned about the local value without knowing all the shares. Function RANDOM(x) returns a uniformly distributed random integer in the range $[0, x-1]$.

The shares calculated this way are then encrypted and added to the corresponding shares, and finally the remaining Sth share is encrypted with the public key of the Sth ancestor and put into the end of the array. This array—that now contains the S shares of the sum of the full sub-tree including the current node—is sent to the parent.

3.3.3 Working with Vectors

We now describe how to efficiently extend our method to vectors of discrete numbers, by packaging multiple elements into a single block of encrypted data. Let us first calculate the number of bits that are required to represent one vector element. Assume that the elements of the input vectors are in the range $[0, m]$. This means that the elements of the output vector fall in range $[0, Nm]$, where N is the mini-batch (tree) size. That is, $M = Nm + 1$. After applying the secret-sharing scheme on an input vector, the elements of the resulting shares also fall in the range $[0, Nm]$ due to the S-out-of-S secret-sharing scheme we apply.

However, when working with homomorphic cryptography, we keep adding encrypted shares together without performing the modulo operation that is required for the correct decoding in our S-out-of-S secret-sharing scheme and for keeping the values in the range $[0, Nm]$. Thus, we need a larger range to accommodate the sum of at most N shares giving us the range of $[0, N^2m]$. This means that $\lceil \log_2(1+N^2m) \rceil$

bits are required per element.

Using this many bits, we can simply concatenate the elements of a share together to form a single bit vector before encryption. Homomorphic addition will result in the corresponding elements being added together. After decryption, the vector can be restored by splitting the bit vector, and element-wise modulo can be performed. This method can be trivially extended to arrays of blocks of a desired size, by packaging the elements into multiple blocks.

3.3.4 Practical Considerations and Optimizations

We stress again that if during the algorithm a child node never responds then its subtree will be essentially missing (will have a sum of zero) but other than that the algorithm will terminate normally. This is acceptable in our application, because for a mini-batch we simply need the sum of any number of gradients, this will not threaten the convergence of the gradient descent algorithm.

The pseudocode discussed above describes a simple and basic version of our algorithm that allows for optimizations to speed up execution. Execution time is important because a shorter execution time allows less time for nodes to fail, in addition, the machine learning algorithm will execute faster as well. A simple optimization is, for example, if, as part of their initialization, all the nodes instantly start encrypting the $S-1$ shares of their local data with the public keys of its $S-1$ closest ancestors.

Another optimization is the parallelization of encryption and sending. Note that encrypting data typically takes much longer than sending it; we will evaluate this in more detail later on. Here, when calculating the message to send to the parent, the node immediately sends the first encoded share to the parent (that is, the share that the parent can decrypt) so that the parent can start working on the decryption. The node then sends all the remaining shares except the Sth share, while calculating its own encryption of the Sth share. Finally, when the encryption is ready, the node sends the Sth share as well.

Also, consider that due to the binomial tree structure, all the leaves are created at about the same time, so they will start to send their message to the parent at about the same time resulting in a more or less round-based aggregation protocol. This makes the time complexity of one such aggregation round in which the aggregation moves up one level (starting from the leaves) $E+T+L$, where E is the encryption/decryption time of a share, T is the transmission time of an encrypted share, and L is the network latency (assuming $E+T>ST$ and that the cost of homomorphic addition is negligible). Note that the actual algorithm does not rely on the existence of synchronized aggregation rounds, in fact, in realistic environments these rounds often overlap if, for example, a node finishes sooner due to losing its children. The rounds are merely an emergent property in reliable environments, a side-effect of using binomial trees as our tree topology.

Another possibility for optimization is based on the observation that shares that would be encrypted with the public keys of the ancestors of the root do not need to be encrypted at all, therefore the root in fact performs only a single decryption.

3.3.5 Variants

Apart from optimizations, one can consider slightly modified versions of the algorithm that can be useful for trading off security and robustness or that allow for a minimal involvement of a central server.

The first variation—that we will actually utilize during our evaluation in Section 3.6—is setting a lower bound on the size of the subtree that we accept. Indeed, we have to be careful when publishing a sum based on too few participants. Let us denote by R the minimal required number of actual participants ($S \leq R \leq N$). Let the nodes pad their messages with an (unencrypted) integer n indicating the number of nodes its data is based on. When the node exactly $S-1$ steps away from the root (thus in the trunk) is about to send its message, it checks whether $n+S-1 \geq R$ holds (since the remaining nodes towards the root have no children except the one on this path). If not, it sends a failure message instead. The nodes fewer than $S-1$ steps away from the root transmit a failure message if they receive one, or if they fail to receive any messages. This way, no nodes can decode the sum of a set that is not large enough.

On a different issue: one can ask the question whether the trunk is needed, as the protocol can be executed on any tree unmodified. However, having no trunk makes it easier to steal information about subtrees close to the root. If the tree is well-balanced and the probability of failure is small, these subtrees can be large enough for the stolen partial sums to not pose a practical privacy problem in certain applications. The advantages include a simpler topology, a faster running time, and increased robustness.

Another option is to replace the top $S-1$ nodes with a central server. To be more precise, we can have a server simulate the top $S-1$ nodes with the local values of these nodes set to zero. This server acts as the root of a 2-trunked tree. From a security point of view, if the server is corrupted by a semi-honest adversary, we have the same situation when the top $S-1$ nodes are corrupted by the same adversary. As we have shown in Section 3.4.1, one needs to corrupt at least S nodes in a chain to gain any extra advantage, so on its own the server is not able to obtain extra information other than the global sum. Also, the server does not need more computational capacity or bandwidth than the other nodes. This variation can be combined with the size propagation technique described above. Here, the child of the server can check whether $n \geq R$ holds.

3.4 Analysis

We first consider the level of security that our solution provides, and we also characterize the complexity of the algorithm.

3.4.1 Security

To steal information, that is, to learn the sum over a subtree, the adversary needs to catch and decrypt all the S shares of the corresponding message that was sent by the root of the subtree in question. Recall that if the adversary decrypts less than S shares from any message, it still has only a uniform random value due to our construction. To be more precise, to completely decrypt a message sent to node c_1, the adversary needs to corrupt c_1 and all its $S-1$ closest ancestors, denoted by $c_2, .., c_S$, so he can obtain the necessary private keys.

The only situation when the shares of a message are not encrypted with the public keys of S *different* nodes—and hence when less than S nodes are sufficient to be corrupted—is when the distance of the sender from the root is less than S. In this case, the sender node is located in the trunk of the tree. However, decrypting such a message does not yield any more information than what can be calculated from the (public) result of the protocol and the local values (gradients) of the nodes needed to be corrupted for the decryption. This is because in the trunk the sender of the message in question is surely the only child of the first corrupted node, and the message represents the sum of the local values of all the nodes, except for the ones needed to be corrupted. To put it in a different way, corrupting less than S nodes never gives more leverage than learning the private data of the corrupted nodes only.

Therefore, the only way to steal extra information (other than the local values of the corrupted nodes) is to form a continuous chain of corrupted nodes $c_1, .., c_S$ towards the root, where c_{i+1} is the parent of c_i. This makes it possible to steal the partial sums of the subtrees rooted at the children of c_1. For this reason we now focus only on the $N-S$ vulnerable subtrees not rooted in the trunk.

As a consequence, a threshold adversary cannot steal information if he corrupts at most $S-1$ nodes. A probabilistic adversary that corrupts each node with probability p can steal the exact partial sum of a given subtree whose root is not corrupted with probability p^S.

Even if the sum of a given subtree is not stolen, some information can be learned about it by stealing the sums of other subtrees. However, this information is limited, as demonstrated by the following theorem.

Theorem 1. *The private value of a node that is not corrupted cannot be exactly determined by the adversary as long as at least one of the S closest ancestors of the node is not corrupted.*

Proof. Let us denote by t the target node, and by u the closest ancestor of t that is not corrupted. The message sent by t cannot be decrypted by the adversary, because one of its shares is encrypted to u (because u is one of the S closest ancestors of t). The same holds for all the nodes between t and u. Therefore the smallest subtree that contains t and whose sum can be stolen also contains u. Due to the nested nature of subtrees, bigger subtrees that contains t also contains u as well. Also, any subtree that contains u also contains t (since t is the descendant of u). Therefore u and t cannot be separated. Even if every other node is corrupted in the subtree whose sum is stolen, only the sum of the private values of u and t can be determined. □

Therefore p^S is also an upper bound on the probability of stealing the exact private value of a given node that is not corrupted.

3.4.2 Complexity

In a tree with a maximal branching factor of B each node sends only one message, and receives at most B. The length of a message (which is an array of S encrypted integers) is $\mathcal{O}(SC)$, where C is the length of the encrypted form of an integer. Let us now elaborate on C. First, as stated before, the sum is represented on $\mathcal{O}(\log M)$ bits, where M is a design choice defining the precision of the fixed point representation of the real values. Let us assume for now that we use the Paillier cryptosystem [66]. In this case, we need to set the parameters of our cryptosystem in such a way that the largest number it can represent is no less than $n = \min(B^S M, NM)$, which is the upper bound of any share being computed by the algorithm (assuming $B \geq 2$). In the Paillier cryptosystem the ciphertext for this parameter setting has an upper bound of $\mathcal{O}(n^2)$ for a single share. Since

$$S \log n^2 = S \log \min(B^S M, NM)^2 \leq 2(S^2 \log B + S \log M), \tag{3.1}$$

the number of bits required is $\mathcal{O}(S^2 \log B + S \log M)$.

The computational complexity is $\mathcal{O}(BSE)$ per node, where E is the cost of encryption, decryption, or homomorphic addition. All these three operations boil down to one or two exponentiations in modular arithmetic in the Paillier cryptosystem. Note that this is independent of N.

The time complexity of the protocol is proportional to the depth of the tree. If the tree is balanced, this results in $S + \mathcal{O}(logN)$ steps altogether.

3.5 Compressing the Gradient

As mentioned in Section 3.3.2, it is essential that we compress the gradient because in a realistic machine learning problem there are at least a few hundred parameters, often a lot more. Encoding and decoding this many floating point numbers with full precision can be prohibitively expensive for our protocol, especially on a mobile device. For this reason, we evaluated the effect of gradient compression on the performance of gradient descent learning. Similar techniques have been used before in a slightly different context [50].

Let us first introduce the exact algorithms and learning tasks we used for this evaluation. As for the learning tasks, we used two data sets. The first is the Spambase binary classification data set from the UCI repository[55], which consists of 4601 records with 57 features. Each of these records belongs to an email that was classified either as spam or as a regular email. The features that represent a piece of email are based on, for example, word and character frequencies or the length of capital letter sequences within the email. 39.4% of the records are positive examples. 10% of the records were reserved for testing. The second dataset we used was based on Reuters articles.[1] It contains 1000 positive and 1000 negative examples, with 600 additional examples used for testing. The examples have 9947 features. The dataset contains Reuters articles and the task is to decide whether a given document is about "corporate acquisitions" or not. The documents are represented by word stam feature vectors, where each feature corresponds to the occurence of a word. Hence, the representation is very high-dimensional and sparse (that is, each vector contains mostly zeros).

We tested two machine learning algorithms. The first is logistic regression [9]. Based on Equation 2.5 and using $\eta_t = \eta/(t+1)$ and $\lambda = 1/\eta$, we used the L2-regularized logistic regression online update rule

$$w \leftarrow \frac{t}{t+1}w + \frac{\eta}{t+1}(f_w(x) - y)x \tag{3.2}$$

where w is the weight vector of the model, t is the number of samples seen by the model (not including the new one), x is the feature vector of the training example, y is the correct label (1 or 0), $f_w(x)$ is the prediction of the model (probability of the label being 1), and η is the learning parameter. We generalize this rule to mini-batches of size E as follows:

$$w \leftarrow \frac{t}{t+E}w + \frac{\eta}{t+E}\sum_{i=1}^{E}(f_w(x_i) - y_i)x_i \tag{3.3}$$

where $(f_w(x_i) - y_i)x_i$ is supposed to be calculated by the individual nodes, and summed using Algorithm 3.1. After the update, t is increased by E instead of 1. η was set to 10^5. The second algorithm was linear SVM [75]. The setup is very

[1]`http://download.joachims.org/svm_light/examples/example1.tar.gz`

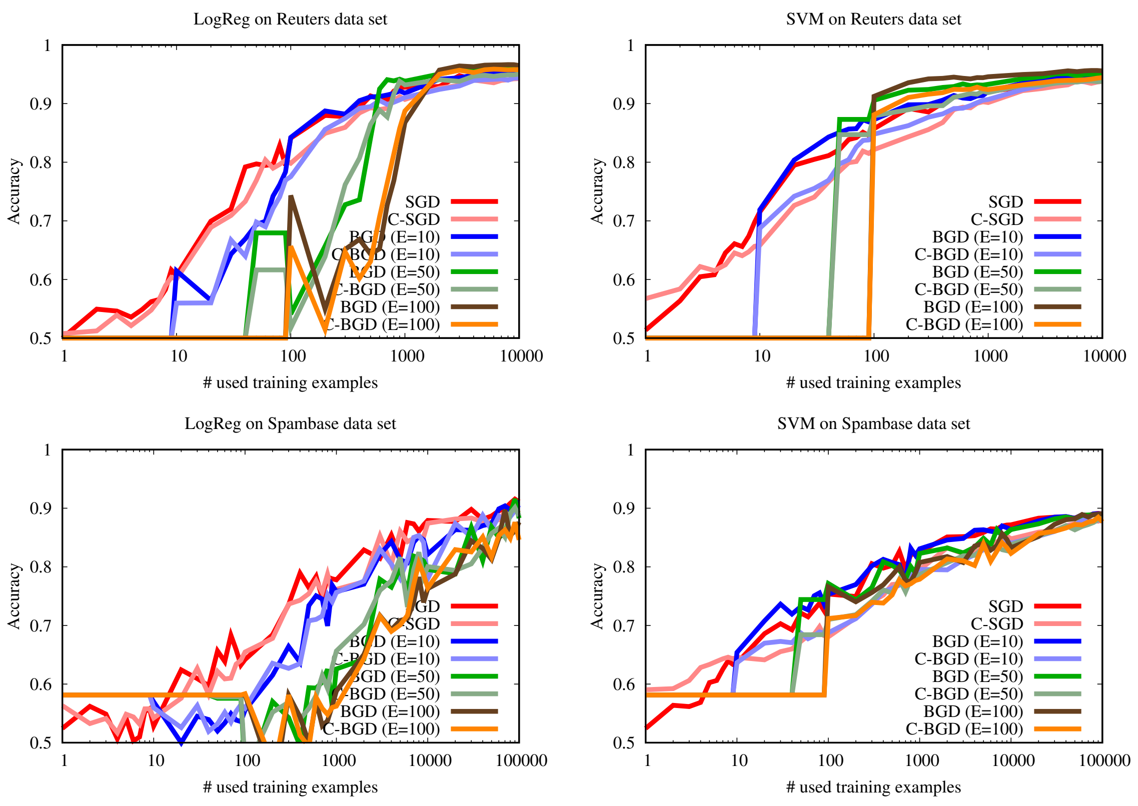

Figure 3.1: *Classification accuracy of the compressed gradient update on the data sets with various batch sizes.*

similar to that of logistic regression, only the batch update rule we used is

$$w \leftarrow \frac{t}{t+E} w + \frac{\eta}{t+E} \sum_{i=1}^{E} [y_i w^T x_i < 1] y_i x_i, \tag{3.4}$$

where $[\cdot]$ is the Iverson bracket notation (1 if its parameter is true, otherwise 0). Here y is the correct label as before, however, now $y \in \{-1, +1\}$.

The compression method we used was the following. All the individual gradients within the mini-batch were computed using a 32-bit floating point representation. These gradients were then quantized by mapping each attribute to one of only three possible values: 1, 0 and -1. This mapping was achieved by stochastic quantization. The quantized value requires only 2 bits to encode, a dramatic compression compared to the original floating point representation of 32 bits. In fact, since we have only three levels, theoretically only a trit is needed for the encoding. We exploit this fact when summing the gradients: the upper bound of the sum of trits (represented on two bits) is lower than the sum of two-bit values. These compressed gradients were then used in equations (3.3) and (3.4) where no further compression is applied.

We ran experiments with all the four possible combinations of learning algorithms and datasets, using four different batch sizes: $E = 1$, 10, 50, and 100. The results are shown in Figure 3.1. The figure shows how the classification accuracy evolves as a function of the number of training examples seen. Accuracy is the proportion of correctly classified instances, that is, the sum of the number of the true positive and the true negative test examples divided by the size of the test set. The databases are well-balanced with respect to the class labels, making this metric adequate. The compressed versions are indicated by the "C-" prefix. It is clear that in these experiments there is virtually no difference between the compressed and original versions. This result is quite striking, and is probably explained by the fact that mini-batch gradients still contain a lot of noise compared to the full gradient even if they are computed exactly.

In the following, we assume that gradient attributes can be safely encoded in two bits only.

3.6 Experimental Evaluation

In this section, our goal is to demonstrate that the decentralized secure mini-batch gradient search we proposed is practically viable, that is, the running time in a real system with realistic parameters is acceptable and that the learning algorithm offers good performance under realistic failure conditions.

Recall that the solution we proposed consists of three components. The first is the overlay tree building algorithm, which defines the mini-batches. The second is the secure sum computation algorithm, which assumes that an overlay tree is given. The third is the applied machine learning algorithm. These three components are modular, different solutions for any of these components can be combined.

We exploit this modularity in our experimental evaluation. First, for each scenario we determine the time that is needed to encrypt and decrypt the messages defined by our secure sum protocol based on the Paillier cryptosystem. We then plug these values into a simulation of the tree building and aggregation protocols under realistic network and failure conditions. The end result of this simulation is a series of mini-batch sizes that are defined by the effective tree-sizes we observe, along with a time-stamp for each mini-batch that depends on the simulated duration of the secure mini-batch gradient computation. Finally, we use these series of mini-batch sizes as well as their timing to assess the performance of the machine learning algorithm in our system. This is possible, because the only important factor for machine learning is the effective size of the tree in each step. We assume that each tree defines a uniform random subset, which is a good approximation if the underlying overlay network is random.

To model the network required for simulating the tree building protocol, we used a real trace of smartphone user behavior [6]. The rest of the parameters defining the computational cost and network utilization were set based on realistic examples. We used PeerSim [59] for our simulations.

3.6.1 Time Consumption

As mentioned above, we first describe the time consumption of the most important operations in our protocol. In order to do that, we carefully have to consider the size of each message that is transmitted and the time needed for encrypting and decrypting these messages. We performed these calculations in a number of scenarios with different parameters that represent interesting use cases. The different scenarios as well as the corresponding message sizes and the amount of time needed to complete a number of different tasks are shown in Table 3.1. In the following we explain these scenarios and the computed values within these scenarios in detail.

For all the trees that we would like to build we fix $S = 4$, as indicated in the first column. This is our security parameter, introduced in Section 3.3.1. The value of $S = 4$ represents a good tradeoff between efficiency and the offered level of security. The binomial tree parameter D (the number of rounds used to build the tree) was set to 4 or 6, giving us the maximum tree sizes of 19 and 67, computed by the formula $N = 2^D + S - 1$, which was explained in detail in Section 3.3.1. The motivation for these settings is that our preliminary experiments with our machine learning application indicated that increasing the mini-batch size beyond 67 is not beneficial. The lower value of 19 is motivated by the fact that smaller trees do not offer a sufficient level of privacy, since the sum is computed based on too few nodes. Also, in a very small tree, the trunk represents a considerable proportion of the tree which limits the possibilities for parallelization, hence the efficiency is not ideal.

The number of features in the learning problem was modeled to be 100 or 10,000. This setting accommodates the number of features in our datasets that are 57 for the Spambase dataset and 9947 for the Reuters dataset (see Section 3.5). Note that we rounded the number up to the closest power of 10 so that we have a 100 times scaling factor, which makes comparison more intuitive.

Based on the tree size N and the quantization parameter m we can compute the number of bits (b) needed to represent a share of one element of the secret-shared gradient vector. As explained in Section 3.3.3 in detail, the formula is given by $b = \lceil \log_2(1 + N^2 m) \rceil$. We used $m = 2$ based on our results on compressing the gradient vector in Section 3.5. The next column shows the key size (or block size) n, a parameter for the Paillier cryptosystem that defines the level of security. We examine the common values 1024 and 2048. Note that 2048 is currently recommended for

Table 3.1: *The used parameter setups, the time consumption of the protocol, and the ratio of good trees.*

Parameter setups								Time consumption (seconds)					Results
S	number of features (f)	D	max tree size (N)	bits per feature (b)	key size (n)	blocks per share $\lceil \frac{fb}{n} \rceil$	message size to parent $S2n\lceil \frac{fb}{n} \rceil$	encrypt / decrypt a block	send plain-text model	encrypt $S-1$ shares	one aggre-gation round	overall time of mini-batch	ratio of good trees
4	10^2	4	19	10	1024	1	8192	0.041	0.103	0.123	0.143	1.847	0.999
					2048	1	16384	0.300	0.103	0.900	0.404	4.451	0.997
		6	67	14	1024	2	16384	0.041	0.103	0.246	0.186	2.850	0.997
					2048	1	16384	0.300	0.103	0.900	0.404	5.466	0.996
	10^4	4	19	10	1024	99	811008	0.041	0.420	12.177	4.362	45.649	0.969
					2048	50	819200	0.300	0.420	45.000	15.305	155.074	0.904
		6	67	14	1024	137	1122304	0.041	0.420	16.851	5.998	74.609	0.951
					2048	69	1130496	0.300	0.420	62.100	21.083	255.624	0.850

sufficient security[2].

Based on the parameters we already defined, we can now compute the number of blocks to be encoded per gradient share: $\lceil \frac{fb}{n} \rceil$. Finally, let us compute the message size to be sent by a node in the tree to its parent. According to the protocol, this message is composed of the S encrypted shares of the compressed gradient. The size of the message is $S2n\lceil \frac{fb}{n} \rceil$ bits. This is due to the fact that the size of an encrypted block is $2n$, and we need $\lceil \frac{fb}{n} \rceil$ blocks per share.

We have now computed almost all the values necessary to determine the time consumption of some important operations of the protocol. The last bit of information required for that is the time consumption of encoding a single block. The Paillier encryption and decryption time of a block is experimentally measured using an unoptimized Java implementation based on BigIntegers on a real Android device (Samsung SM-T280). This can be considered a worst case scenario because the implementation we used has a lot of room for optimization and the device itself is not an up-to-date model. Both the encryption and decryption take 0.041 s with a 1024 bit key and 0.300 s with a 2048 bit key.

Sending the model in plaintext from the parent to the child is required when building the tree. We assume single precision floating-point arithmetic (32 bits) so the sizes of the linear models are 3,200 bit and 320,000 bit for 100 and 10,000 features, respectively. The actual sending time is given by the 1 Mbps bandwidth we allow between online nodes and assuming a 100 ms latency. After receiving the model in plaintext the node instantly starts encrypting $S-1$ shares as discussed in Section 3.3.4. This takes $S-1$ times the encryption time of all the required blocks. The computed values are shown in Table 3.1.

The next column shows the time of one aggregation round, that is, the time needed for a child node to propagate information up to the parent. In Section 3.3.4 we described a number of variants of the protocol that involve different optimizations compared to the basic variant. Here, we assume the variant, in which children in the tree start encrypting their share while they simultaneously upload the other $S-1$ shares to their parents. In all our scenarios uploading $S-1$ shares is faster than encrypting one share. This means that the time needed for one aggregation round is the time of encoding one share plus the time of uploading this share (which consists of transmission time and network latency). The column indicating the time needed for one aggregation round shows this value for each parameter setting.

The column that corresponds to the overall mini-batch time sums up all the required times for completing the mini-batch, assuming the network is error free. This involves sending the plaintext model to the children down the tree during tree building as well as the aggregation rounds up to the root. These operations are performed for each level of the tree; note that the depth of the whole tree is $D+S-1$. The time

[2] `https://www.keylength.com/`

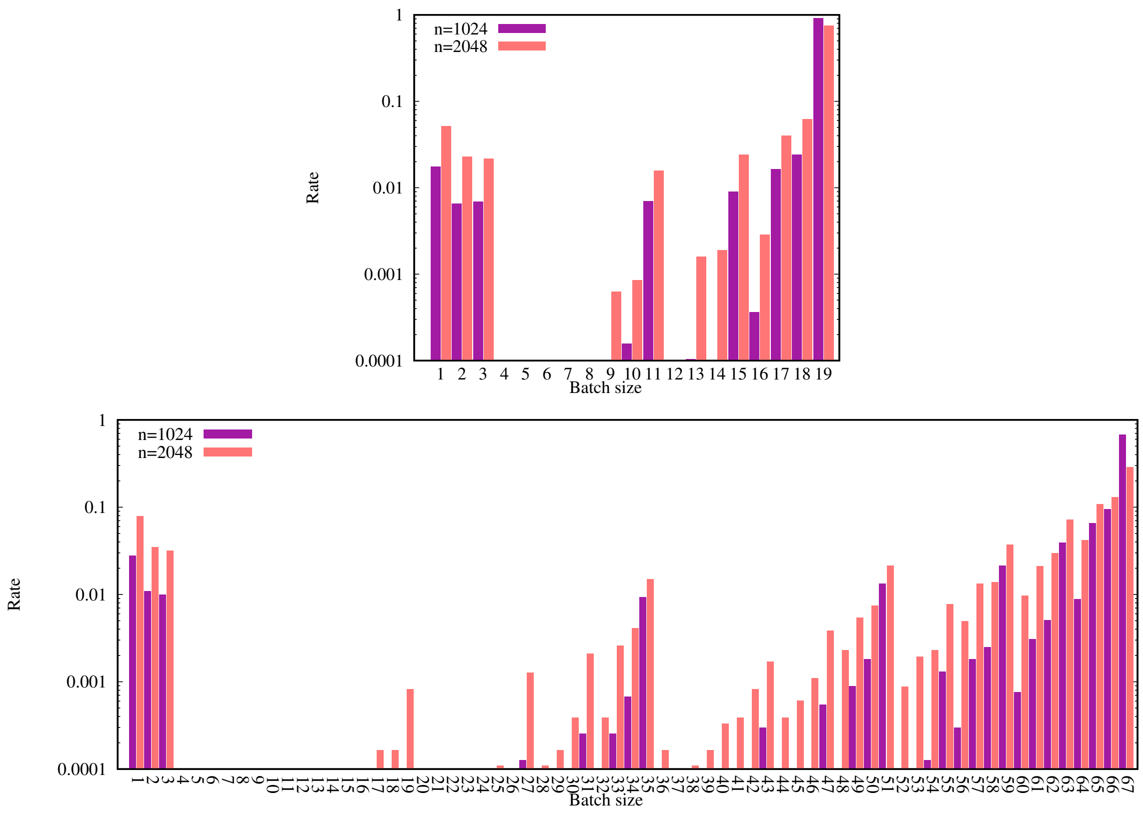

Figure 3.2: *Distribution of effective mini-batch sizes for scenario of 10,000 features. The histograms use a logarithmic scale.*

of encoding $S-1$ shares also needs to be added because the leaves must first complete this encoding before starting the first aggregation round. If nodes can fail, in an actual run these times may be slightly longer because of the delay introduced by the failure detector, but they may also be slightly shorter, due to a smaller tree. Our simulations account for these effects. Note that we ignored the time consumption of the single gradient update step that has to be performed as well at every node. This is because the encryption operation is orders of magnitude slower than the gradient update.

3.6.2 Simulating Tree Building

All of our experiments were run on top of the churn trace described in Section 2.4, except that we used one-day segments. However, the first 10 seconds of each online session are considered offline because extremely short online sessions would introduce unreliability. This technique can also be explicitly implemented as part of our protocol: a node should simply wait 10 seconds before joining the network.

The network size was 100,000. The membership overlay network was implemented by independently assigning 100 randomly selected outgoing neighbors to each node and then dropping the directionality of the links. This network forms the basis of tree building, the tree neighbors are selected from these nodes. We assume that each node maintains an active TCP connection with its neighbors as suggested in [71]. If a node fails, its neighbors will detect this only with a one second delay. The neighbor set is constant in our simulations; that is, when a neighbor fails it remains on the list and it is reconnected when it comes back online. The size of our neighbor set was large enough for the overlay network to remain connected.

Initially a random online node is picked from the network at time 0:00 and we simulate building the first tree using that node as root. This simulation involves building the tree and propagating the aggregated gradient up to the root, simulated based on the time consumption of these operations described previously. When this is completed, we pick a new random node that is online at the time of finishing the first mini-batch and simulate a new mini-batch round. We repeat this procedure until the end of the simulated day. With this methodology, we record the effective mini-batch sizes (which determines the number of gradients the sum of which the root actually received) and we examine the distribution of these effective mini-batch sizes.

The empirical distributions of the effective mini-batch sizes for the case of 10,000 features are shown in Figure 3.2. In every scenario we simulated a sample of at least 15,000 tree building attempts. The figure shows the histograms based on these samples. The histograms use a logarithmic scale to better illustrate the structure of the distribution. However, note that most of the probability mass belongs to the largest effective sizes. For 100 features almost all the trees are complete due to the very quick building times (not shown). The relatively high probability mass for tree sizes 1, 2 and 3 are due to the vulnerability of the trunk.

In our experiments, we used the variant of the protocol that limits the effective tree size from below as explained in Section 3.3.5. We accepted a mini-batch for gradient update only if its size was greater than or equal to $\lfloor \frac{N}{2} \rfloor$. The reason is that smaller trees represent reduced privacy. We call such trees a “good tree”. The last column of Table 3.1 contains the probability of getting a good tree. Clearly, only a very small proportion of tree building attempts are unsuccessful.

3.6.3 Machine Learning Results

We now present our results with the actual learning tasks. The setup for the learning problems is identical to that presented in Section 3.5. The only difference is that now the batch sizes used in each update step are variable and depend on the effective batch size that is obtained in our tree building simulation based on the smartphone trace (assuming one example per node), and the time needed to complete a given mini-batch is also given by the output of the simulation. The results are shown in

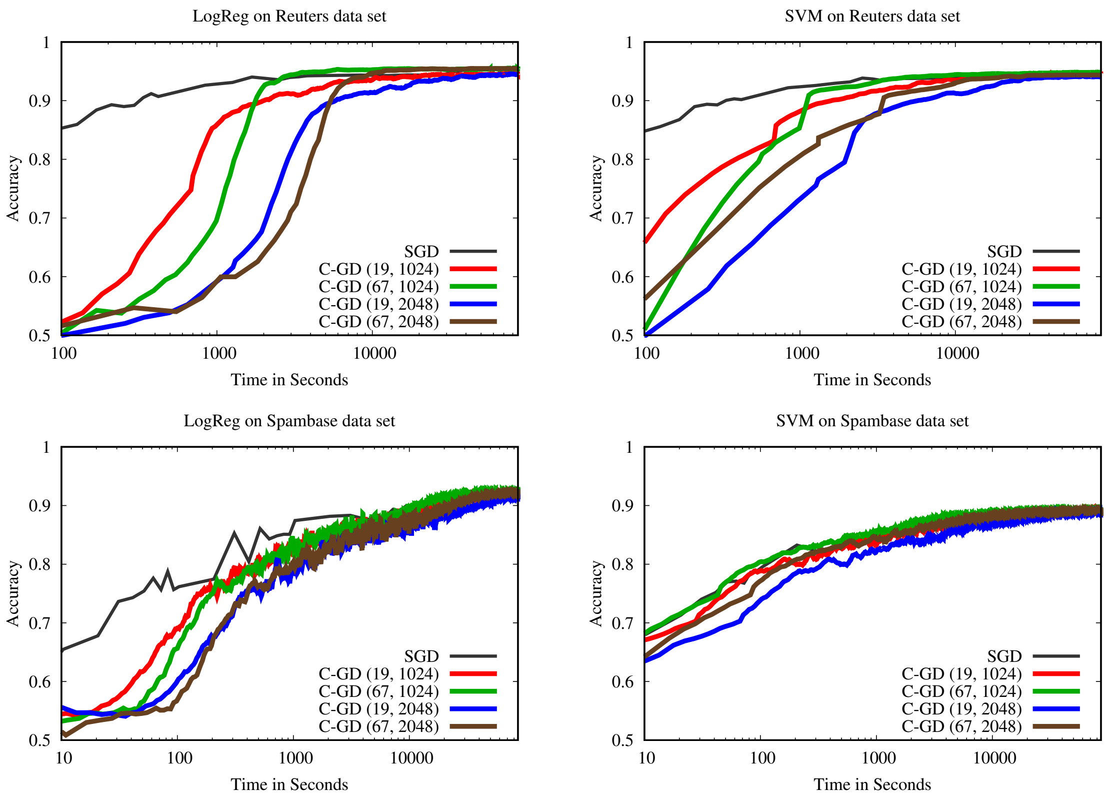

Figure 3.3: *Classification accuracy of the compressed gradient update on the data sets based on trace-based simulation. We vary key size (1024 or 2048) and maximum tree size (19 or 67).*

Figure 3.3. Note that the horizontal axis of the plots now shows the time, covering one full day. It is clear that the main factor for convergence speed is the encryption key size, with 2048 being significantly slower than 1024. This could be expected based on Table 3.1 as well. We can see that our example learning tasks can converge within one day, which is adequate for many practically interesting learning problems.

3.7 Conclusion

We proposed a secure sum protocol to prevent the collusion attack in gossip learning. The main idea is that instead of SGD we implement a mini-batch method and the sum within the mini-batch is calculated using our novel secure algorithm. We can achieve high levels of robustness and good scalability in our tree building protocol through exploiting the fact that the mini-batch gradient algorithm does not require the sum to be precise. The algorithm runs in logarithmic time and it is designed to calculate a partial sum in case of node failures. It can tolerate collusion unless there are S

consecutive colluding nodes on any path to the root of the aggregation tree, where S is a free parameter. The algorithm is completely local therefore it has the same time-complexity independently of the network size.

We evaluated the protocol in realistic simulations where we took into account the time needed for encryption and message transmission, and we used a real smartphone trace to simulate churn. We demonstrated on a number of learning tasks that the approach is indeed practically viable even with a key size of 2048. We also demonstrated that the gradients can be compressed by an order of magnitude without sacrificing prediction accuracy.

Contribution

In this chapter, the contributions of the author were: a scalable and robust secure sum protocol that is able to securely compute a partial sum even in the event of failures and limited collusion of nodes; a proof about its capability of preventing the collusion attack; and a decentralized mini-batch gradient descent method based on the building of a k-trunked binomial overlay tree and the above protocol. Árpád Berta performed the detailed empirical evaluation of the proposed overlay tree building using a smartphone churn trace.

Algorithm 3.1 Robust secure sum

```
procedure INIT
    shares ← new array[$1..S$]
    for $i \leftarrow 1$ to $S$ do
        shares[$i$] ← Encrypt(0, Ancestor($i$))
    end for
    knownShare ← $0$
end procedure

procedure ONMESSAGERECEIVED(msg)
    for $i \leftarrow 1$ to $S - 1$ do
        shares[$i$] ← shares[$i$] $\oplus$ msg[$i + 1$]
    end for
    knownShare ← knownShare + Decrypt(msg[1])
end procedure

procedure ONNOMOREMESSAGESEXPECTED
    if IAmTheRoot() then
        for $i \leftarrow 1$ to $S - 1$ do
            knownShare ← knownShare + Decrypt(shares[$i$])
        end for
        Publish((knownShare + localValue) $\bmod M$)
    else
        randSum ← $0$
        for $i \leftarrow 1$ to $S - 1$ do
            rand ← Random($M$)
            randSum ← randSum + rand
            shares[$i$] ← shares[$i$] $\oplus$ Encrypt(rand, Ancestor($i$))
        end for
        knownShare ← knownShare + localValue − randSum
        shares[$S$] ← Encrypt(knownShare $\bmod M$, Ancestor($S$))
        SendToParent(shares)
    end if
end procedure
```

Chapter 4

Comparison of Federated and Gossip Learning

Performing data mining over data collected by edge devices, most importantly, mobile phones, is of great interest [82]. Collecting such data at a central location has become more and more problematic in the past few years due to novel data protection rules [32] and in general due to the increasing public awareness of issues related to data handling. For this reason, there is an increasing interest in methods that leave the raw data on the device and process it using distributed aggregation.

Google introduced *federated learning* to answer this challenge [50, 58]. This approach is very similar to the well-known parameter server architecture for distributed learning [24] where worker nodes store the raw data. The parameter server maintains the current model and regularly distributes it to the workers who in turn calculate a gradient update and send it back to the server. The server then applies all the updates to the central model. This is repeated until the model converges. In federated learning, this framework is optimized so as to minimize communication between the server and the workers. For this reason, the local update calculation is more thorough, and compression techniques can be applied when uploading the updates to the server.

In addition to federated learning, *gossip learning* has been proposed to address the same challenge [38, 65]. This approach is fully decentralized, no parameter server is necessary. Here, nodes exchange and aggregate models directly. The advantages of gossip learning are obvious: since no infrastructure is required, and there is no single point of failure, gossip learning enjoys a significantly *cheaper scalability and better robustness*. A key question, however, is how the two approaches compare in terms of performance. This is the question we address in this chapter.

We compare the two approaches in terms of convergence time and model quality, assuming that both approaches utilize the same amount of communication resources in the same scenarios. In other words, we are interested in the question of whether—by communicating the same number of bits in the same time-window–the two approaches can achieve the same model quality. We train linear models using stochastic gradient descent (SGD) based on the logistic regression loss function.

Our experimental methodology involves several scenarios, including smartphone churn traces collected by the application Stunner [6]. We also vary the network size. In addition, we evaluate different assumptions about the label distribution; that is, whether a given worker has a biased or unbiased subset of the training samples.

To make the comparison as fair as possible, we ensure that the two approaches differ mainly in their communication patterns. However, the computation of the local update is identical in both approaches. Also, we apply subsampling to reduce communication in both approaches, as introduced in [50] for federated learning. Here, we adapt the same technique for gossip learning.

We note that both approaches offer mechanisms for explicit privacy protection, apart from the basic feature of not collecting data. In federated learning, Bonawitz et al. [11] describe a secure aggregation protocol, whereas for gossip learning one can apply the methods described in [19]. Here, we are concerned only with the efficiency of the different communication patterns and do not compare security mechanisms.

The result of our comparison is that gossip learning is in general comparable to the centrally coordinated federated learning approach. This result is rather counter-intuitive and suggests that decentralized algorithms should be treated as first class citizens in the area of distributed machine learning overall, considering the additional advantages of decentralization.

The outline is as follows. In sections 4.1 and 4.2 we describe gossip learning and federated learning, respectively. We describe our novel algorithms as well, including the the model partitioning technique, and a number of minor design decisions that allow all the evaluated algorithms to use shared components. In Section 4.3, we present our experimental setup that includes the datasets used, as well as our system model. We also discuss the problem of choosing hyperparameters. In Section 4.4, we describe our experimental results for a range of scenarios with many algorithm variants. In Section 4.5, we present related work, then in Section 4.6 we draw our conclusions.

4.1 Gossip Learning

We discussed the basic notions of gossip learning in Section 2.3. Here, we describe the variant with sampling capability.

Each node k runs Algorithm 4.1. First, the node initializes its local model (w_k, b_k) and its age t_k. A subset of the model parameters (along with the model age) is then periodically sent to another node in the network. When a node receives such a parameter sample, it merges it into its own model and then it performs a local update

Algorithm 4.1 Gossip Learning with Sampling

```
 1: $(t_k, w_k, b_k) \leftarrow (\mathbf{0}, \mathbf{0}, 0)$
 2: loop
 3:     wait($\Delta_g$)
 4:     $p \leftarrow$ selectPeer()
 5:     send sample($t_k, w_k, b_k$) to $p$
 6: end loop
 7:
 8: procedure onReceiveModel($t_r, w_r, b_r$)
 9:     $(t_k, w_k, b_k) \leftarrow$ merge($(t_k, w_k, b_k), (t_r, w_r, b_r)$)
10:     $(t_k, w_k, b_k) \leftarrow$ update($(t_k, w_k, b_k), D_k$)
11: end procedure
```

step. Note that the rounds are not synchronized, although all the nodes use the same period Δ_g. Any received messages are processed immediately. Different variants of the algorithm can be produced with different implementations of the methods called SAMPLE, MERGE, and UPDATE. In the simplest case, SAMPLE sends the entire model (no sampling), MERGE computes the average, and UPDATE performs a mini-batch update based on the local data. Below in Section 4.1.1 we shall define more sophisticated implementations.

The node selection in line 4 is supported by a so-called peer sampling service. Applications can utilize a peer sampling service implementation to obtain random samples from the set of participating nodes. The implementations of this service might be based on several different approaches that include random walks [78], gossip [45], or even static overlay networks that are created at random and repaired when necessary [71]. We will assume a static, connected, random overlay network from now on.

In the following, we shall describe optimizations of the original gossip learning algorithm. Stated briefly, the basic ideas behind them are the following:

Sampling: Instead of sending the full model to the neighbor, a node can send only a subset of the parameters. This technique is often used as a compression mechanism to save bandwidth.

Model partitioning: Related to sampling, instead of a random subset, it is also possible to define a fixed partitioning of the model parameters and to send one of these subsets as a sample.

Let us now discuss these techniques in turn.

Algorithm 4.2 Partitioned Model Merge

```
1: $S$ : the number of partitions
2: procedure MERGE($(t, w, b), (t_r, w_r, b_r)$)
3:     $j \leftarrow$ index of received partition    ▷ $j = i \mod S$, for any coordinate $i$ within the sample
4:     for coordinate $i$ is included in sample do
5:         $w[i] \leftarrow (t[j] \cdot w[i] + t_r[j] \cdot w_r[i])/(t[j] + t_r[j])$
6:     end for
7:     $b \leftarrow (t[S] \cdot b + t_r[S] \cdot b_r)/(t[S] + t_r[S])$
8:     $t \leftarrow \max(t, t_r)$    ▷ element-wise maximum, where $t_r$ is defined
9:     return $(t, w, b)$
10: end procedure
```

4.1.1 Random Sampling and Model Partitioning

As for the method SAMPLE, we will use two different implementations. The first implementation, SAMPLERANDOM(t, w, b, s) returns a uniform random subset of the parameters, where $s \in (0, 1]$ defines the size of the sample. To be precise, the size of the sample is given by $s \cdot d$ (randomly rounded), where d is the dimension of the vector w.

The other implementation is based on a partitioning of the model parameters. Let us elaborate on the idea of model partitioning here. The model is formed by the vector w and the bias value b. We partition only w. We define $S \geq 1$ partitions by assigning a given vector index i to the partition index $(i \mod S)$. When sampling is based on this partitioning, we return a given partition. More precisely, SAMPLEPARTITION(t, w, b, j), where $0 \leq j < S$ is a partition index, returns partition j. The bias b is always included in each sample, in both implementations of method SAMPLE.

It is important to stress that the random sampling method SAMPLERANDOM should be applied without model partitioning (that is, when $S = 1$). It is possible to define a combination of partition-based and random sampling, where we could sample from a given partition, but we do not explore this possibility here.

Upon receiving a model, the node merges it with its local model, and updates it using its local data set D_k. Method MERGE is used to combine the local model with the incoming one. The most usual way to implement MERGE is to take the average of the parameter vectors [65]. This has some theoretical justification as well, at least in the case of linear models and when there is only one round of communication [89].

If there is no partitioning ($S = 1$) then the implementation shown as Algorithm 4.2 computes the average weighted by model age. This implementation can handle subsampled input as well, as we consider only those parameters that are actually included in the sample. When partitioning is applied (that is, when $S > 1$),

Algorithm 4.3 Partitioned Model Update Rule

```
1: $S$ : the number of partitions
2: $d$ : the dimension of $w$
3: procedure UPDATE($(t, w, b), D$)
4:     for all batch $B \subseteq D$ do                                   ▷ D is split into batches
5:         $t \leftarrow t + |B| \cdot \mathbf{1}$                                    ▷ increase all ages by $|B|$
6:         for $i \in \{1, ..., d\}$ do
7:             $h[i] \leftarrow -\frac{\eta}{t[i \mod S]} \sum_{(x,y)\in B} (\frac{\partial \ell(f_{w,b}(x), y)}{\partial w[i]}(w[i]) + \lambda w[i])$
8:         end for
9:         $g \leftarrow -\frac{\eta}{t[S]} \sum_{(x,y)\in B} (\frac{\partial \ell(f_{w,b}(x), y)}{\partial b}(b) + \lambda b)$
10:        $w \leftarrow w + h$
11:        $b \leftarrow b + g$
12:    end for
13:    return $(t, w, b)$
14: end procedure
```

each partition of the parameter vector has its own age parameter. This means that every model now has a vector of age values t of length $S + 1$ where the ages of the partitions are $t[0], \ldots, t[S-1]$ and the age of the bias is $t[S]$.

Method UPDATE is shown in Algorithm 4.3. This implementation requires a full model as input, but it does take into account the partitioning of the model in that all the partitions have their own dynamic learning rate that is determined by the age of the partition.

4.2 Federated Learning

Federated learning is not a specific algorithm, but more of a design framework for edge computing. We discuss federated learning based on the algorithms presented in [50, 58]. While we keep the key design elements, our presentation contains small adjustments and modifications to accommodate our contributions, and allow gossip learning and federated learning to share a number of key methods.

The pseudocode of the federated learning algorithm is shown in algorithms 4.4 (master) and 4.5 (worker). The master periodically sends the current model w to all the workers asynchronously in parallel and collects the answers from the workers. In this version of the algorithm, communication is compressed by sampling the parameter vectors. The rate of sampling might be different in the case of downstream messages (line 4 of Algorithm 4.4) and upstream messages (line 11 of Algorithm 4.5). We require that $s_{up} \leq s_{down}$. Although this is not reflected in the pseudocode for

Algorithm 4.4 Federated Learning Master

```
 1: $(t, w, b) \leftarrow$ init()
 2: loop
 3:     for every node $k$ in parallel do          $\triangleright$ non-blocking (in separate threads)
 4:         send sample$(t, w, b, s_{down})$ to $k$
 5:         receive $(n_k, h_k, g_k)$ from $k$      $\triangleright$ $n_k$: #examples at $k$; $h_k$: sampled model gradient; $g_k$: bias gradient
 6:     end for
 7:     wait$(\Delta_f)$                                $\triangleright$ the round length
 8:     $n \leftarrow \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} n_k$          $\triangleright$ $\mathcal{K}$: nodes that returned a model in this round
 9:     $t \leftarrow t + n$
10:     $h \leftarrow$aggregate$(\{h_k : k \in \mathcal{K}\})$
11:     $w \leftarrow w + h$
12:     $g \leftarrow \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} g_k$
13:     $b \leftarrow b + g$
14: end loop
```

presentation clarity, the sample produced in line 11 of Algorithm 4.5 is allowed to include only indices that are also included in the received model. For example, if $s_{up} = s_{down}$ then the worker selects exactly those indices that were received in the incoming sample.

Any answers from workers arriving with a delay larger than Δ_f are simply discarded. After Δ_f time units have elapsed, the master aggregates the received gradients and updates the model. We also send and maintain the model age t (based on the average number of examples used for training) in a similar fashion, to make it possible to use dynamic learning rates in the local learning algorithm.

We note that, while in this version of the algorithm the master sends the model to every worker, it is possible to use a more fine-grained method to select a subset of workers that get the model in a given round. For example, if the workers have a very limited budget of communication, it might be better to avoid talking to each worker in each round. In fact, we will study such a scenario during our experimental evaluation, but we did not want to include this option in the pseudocode for the sake of clarity.

These algorithms are very generic, the key characteristics of federated learning lying in the details of the update method (line 9 of Algorithm 4.5) and the aggregation mechanism (line 10 of Algorithm 4.4). The update method is typically implemented via a minibatch gradient descent algorithm that operates on the local data, initialized with the received model w. The implementation we use here is identical to that of gossip learning, as given in Algorithm 4.3. Note that here we do not partition the model (that is, $S = 1$). As for sampling, we use sampleRandom as described in Section 4.1.1.

Algorithm 4.5 Federated Learning Worker

```
1: $(t_k, w_k, b_k) \leftarrow$ init()    ▷ the local model at the worker
2:
3: procedure onReceiveModel($t, w, b$)    ▷ $w$: sampled model
4:     $t_k \leftarrow t$
5:     for $w[i] \in w$ do    ▷ coordinate $i$ is defined in $w$
6:         $w_k[i] \leftarrow w[i]$
7:     end for
8:     $b_k \leftarrow b$
9:     $(t_k, w_k, b_k) \leftarrow$ update($(t_k, w_k, b_k), D_k$)    ▷ $D_k$: the local database of examples
10:    $(n, h, g) \leftarrow (t_k - t, w_k - w, b_k - b)$    ▷ $n$: the number of local examples, $h$: the gradient update
11:    send sample($n, h, g, s_{up}$) to master
12: end procedure
```

Method AGGREGATE is used in Algorithm 4.4. Its function is to aggregate the received sampled gradients. Possible implementations are shown in Algorithm 4.6. Both implementations are unbiased estimates of the average gradient. This also implies that when there is no actual sampling (that is, we have $s = 1$) then simply the average of the gradients is computed by both methods. The improved version averages each coordinate separately; that is, it takes the average of only those coordinates that are included in the sample. This is a more accurate estimate of the true average of the given coordinate. However, in order to get an unbiased estimate, we have to divide by the probability that there is at least one gradient in which the given coordinate is included. This probability equals $1 - (1 - s)^{|H|}$. Note that this probability is independent of the coordinate i, so its effect can be thought of correcting the learning rate, especially when $|H|$ is small.

4.3 Experimental Setup

4.3.1 Datasets

We used three datasets taken from the UCI machine learning repository [28] to test the performance of our algorithms. The first is the Spambase (SPAM E-mail Database) dataset containing a collection of emails. Here, the task is to decide whether an email is spam or not. The emails are represented by high level features, mostly word or character frequencies. The second dataset is Pendigits (Pen-Based Recognition of Handwritten Digits), which contains downsampled images of 4×4 pixels of digits from 0 to 9. The third is the HAR (Human Activity Recognition Using Smartphones) [2] dataset, where human activities (walking, walking upstairs, walk-

Algorithm 4.6 Variants of the aggregate function

```
1: procedure AGGREGATE($H$)
2:     $h' \leftarrow \mathbf{0}$
3:     for $i \in \{1, ..., d\}$ do
4:         $h'[i] \leftarrow \frac{1}{s|H|} \sum_{h \in H : h[i] \in h} h[i]$        ▷ $s \in (0, 1]$: sampling rate used to create $H$
5:     end for
6:     return $h'$
7: end procedure
8:
9: procedure AGGREGATEIMPROVED($H$)
10:    $h' \leftarrow \mathbf{0}$
11:    for $i \in \{1, ..., d\}$ do
12:        $H_i \leftarrow \{h : h \in H \wedge h[i] \in h\}$
13:        $h'[i] \leftarrow \frac{1}{|H_i|(1-(1-s)^{|H|})} \sum_{h \in H_i} h[i]$        ▷ skipped if $|H_i| = 0$
14:    end for
15:    return $h'$
16: end procedure
```

Table 4.1: *Data set properties*

	Spambase	Pendigits	HAR
Training set size	4140	7494	7352
Test set size	461	3498	2947
Number of features	57	16	561
Number of classes	2	10	6
Class-label distribution	≈ 6:4	≈ uniform	≈ uniform

ing downstairs, sitting, standing and laying) were monitored by smartphone sensors (accelerometer, gyroscope and angular velocity). High level features were extracted from these measurement series.

The main properties, such as size or number of features, are listed in Table 4.1. In our experiments we standardized the feature values; that is, we shifted and scaled them so as to have a mean of 0 and a variance of 1. Note that the standardization can be approximated by the nodes in the network locally if the approximations of the statistics of the features are fixed and known, which can be ensured in a fixed application.

In our simulation experiments, each example in the training data was assigned to one node when the number of nodes was 100. This means that, for example, with the HAR dataset each node gets 73.5 examples on average. The examples were assigned evenly, that is, the number of examples at the nodes differed by at most one due to the number of samples not being divisible by 100. When the network size equaled the database size, we mapped the examples to the nodes so that each node had exactly one example. We also experimented with a third scenario that combines the above two settings. That is, the number of examples per node was the same as in the 100 node scenario, but the network size was the same as the database size. To achieve this, we replicated the examples, that is, each example was assigned to multiple nodes.

In the scenarios where a node had more than one example, we considered two different class label distributions. The first one is *uniform assignment*, which means that we assigned the examples to nodes at random independently of the class label. The second one is *single class assignment* when each node has examples just from a single class. Here, the different class labels are assigned uniformly to the nodes, and then the examples with a given label are assigned to one of the nodes with the same label, uniformly. These two assignment strategies represent the two extremes in any real application. In a realistic setting the class labels will likely be biased but much less so than in the case of the single class assignment scenario.

4.3.2 System Model

In our simulation experiments, we used a fixed random k-out overlay network, with $k = 20$. That is, each node had $k = 20$ fixed random neighbors. As described previously, the network size was either 100 or the same as the database size. In the churn-free scenario, every node stayed online for the whole experiment. The churn scenario is based on a real trace gathered from smartphones (as described in Section 2.4, but with 1-day segments). We assumed that a message is successfully delivered if and only if both the sender and the receiver remains online during the transfer. We also assume that the nodes are able to detect which of their neighbors are online at any given time with a delay that is negligible compared to the transfer time of a model. Nodes retain their state while being offline.

We assumed that the server has unlimited bandwidth. In practice, unlimited bandwidth is achieved using elastic cloud infrastructure, which obviously has a non-trivial cost in a very large system. Gossip learning has *no additional cost at all related to scaling*. Hence, ignoring the cost of the cloud infrastructure clearly favors federated learning in our study, so this assumption should be kept in mind.

We assumed that the worker nodes have identical upload and download bandwidths. This needs explanation, because in federated learning studies, downstream communication is considered free, citing the fact that the available upload bandwidth is normally much lower than the download bandwidth. But this distinction is only relevant if all the nodes are allowed to use their full bandwidth completely dedicated to federated learning continuously. This is a highly unlikely scenario, given that federated learning will not be the only application on most devices. It is much more likely that there will be a cap on the bandwidth usage, in which light the difference between upstream and downstream bandwidth fades. For the same reason, we also assumed that all the worker nodes have the same bandwidth because the bandwidth cap mentioned above can be expected to be significantly lower than the average available bandwidth, so this cap could be assumed to be uniform.

One could, of course, set a higher cap on downstream traffic. Our study is also relevant in this scenario, for two reasons. First, the actual bandwidth values make no qualitative difference if the network is reliable (there is no churn), they result only in the scaling of time. That is, scaling our results accordingly provides the required measurements. Second, in unreliable networks, a higher downstream bandwidth would result in a similar scaling of time. In addition, it would result in better convergence as well, since the downstream messages could be delivered with a strictly higher probability.

In the churn scenario, we need to fix the amount of time necessary to transfer a full model. (If the nodes are reliable then the transfer time is completely irrelevant, since the dynamics of convergence are identical apart from scaling time.) The transfer time of a full model was assumed to be $60 \cdot 60 \cdot 24/1000 = 86.4$ seconds, irrespective of the dataset used, in the *long transfer time* scenario, and 8.64 seconds in the *short transfer time* scenario. This allowed us to simulate around 1,000 and 10,000 iterations over the course of 24 hours, respectively. Note that the actual models in our simulation are relatively small linear models, so they would normally require only a fraction of a second to be transferred. Still, we pretend here that our models are very large. This is because if the transfer times are very short, the network hardly changes during the learning process, so in effect we learn over a static subset of the nodes. Long transfer times, however, make the problem more challenging because many transfers will fail, as in the case of very large machine learning models such as deep neural networks.

4.3.3 Hyperparameters

The goal was to make sure that the protocols communicate as much as they can under the given bandwidth constraint. The gossip cycle length Δ_g is thus exactly the transfer time of a full model; that is, all the nodes send messages continuously. The cycle length Δ_f of federated learning is the round-trip time, that is, the sum of the upload and download transfer times. When compression is used, the transfer time is proportionally less as defined by the compression rate, and this is also reflected in the cycle length settings.The two algorithms transfer the same number of bits overall in the network during the same amount of time.

As for subsampling, we explore the sampling probability values $s \in \{1, 0.5, 0.25, 0.1\}$. In federated learning, both the upstream and the downstream messages can be sam-

Table 4.2: *Hyperparameters*

	Spambase	Pendigits	HAR
Parameter η	10^3	10^4	10^2
Parameter λ	10^{-3}	10^{-4}	10^{-2}

pled using a different rate, denoted by s_{up} and s_{down}, respectively. The setting we use in our experiments is $s_{down} = s_{up}$. However, we shall also show runs where we fix $s_{down} = 1$ and experiment with upstream sampling only. When the subsampling is based on partitioning, the number of partitions S defines the sampling probability, which equals $1/S$. On the plots $s = 1/S$ will be used even in the partitioned case to indicate the compression rate.

We train a logistic regression model. For the datasets that have more than two classes (Pendigits, HAR), we embedded the model in a one-vs-all meta-classifier. The learning algorithm used was stochastic gradient descent. The learning rate η and the regularization coefficient λ used in our experiments are shown in Table 4.2. We used grid search to optimize these parameters in various scenarios, and found these values relatively robust. However, one should bear in mind that including additional hyperparameters in the search, such as the number of iterations (which we simply fixed here), could result in a different outcome.

Although we fixed the parameters shown in Table 4.2 in all the scenarios, it is interesting to have a more fine-grained look at the behavior of these hyperparameters. This sheds some light on possible heuristics to pick the right parameter values. All the examples here are measurements with gossip learning with model partitioning and $S = 10$. With a network size $N = 100$, over the Pendigits dataset, after 10 gossip cycles, the optimal hyperparameters are $\eta = 10^2$ and $\lambda = 10^{-2}$, as shown in Figure 4.1. However, after 100 cycles, the optimal values are $\eta = 10^3$ and $\lambda = 10^{-3}$, and after 1000 cycles, $\eta = 10^4$ and $\lambda = 10^{-4}$. We often observed similar trends also with other algorithms and datasets. Notice that settings where $\eta\lambda = 1$ (points on the diagonal of the grid) are often a good choice; however, there are exceptions. For instance, when we have only one example per node, over the Spambase dataset, this is not the case, as shown in Figure 4.2. As we can see, here, the points above the diagonal tend to be somewhat better. Also note that settings with $\eta\lambda > 1$ (points below the diagonal) tend to perform very poorly.

4.4 Experimental Results

We ran the simulations using PeerSim [59]. As for the hardware requirements for reproducing our results, we used a server with 8 2 GHz CPUs with 8 cores each,

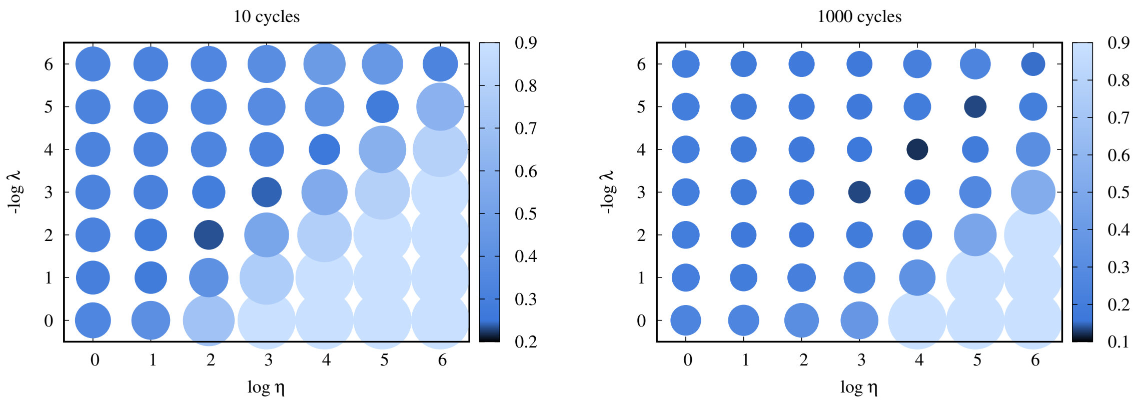

Figure 4.1: *The error of partitioned gossip learning with $S = 10$ and $N = 100$ on the Pendigits dataset as a function of η and λ after 10 cycles (left) and after 1000 cycles (right).*

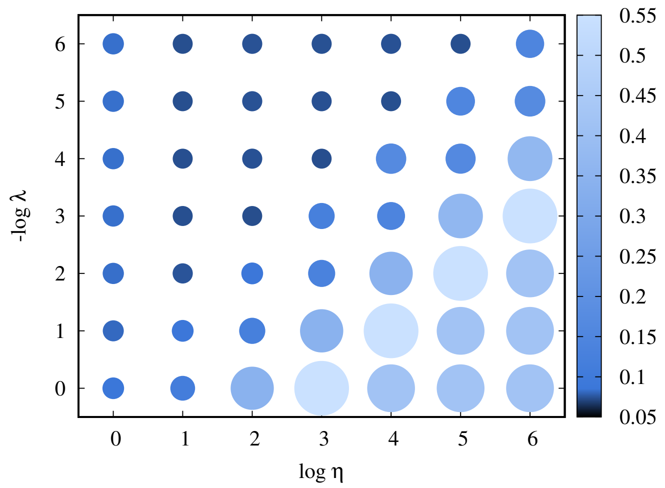

Figure 4.2: *The error of partitioned gossip learning with $S = 10$ and $N = 4140$ on the Spambase dataset as a function of η and λ after 1000 cycles.*

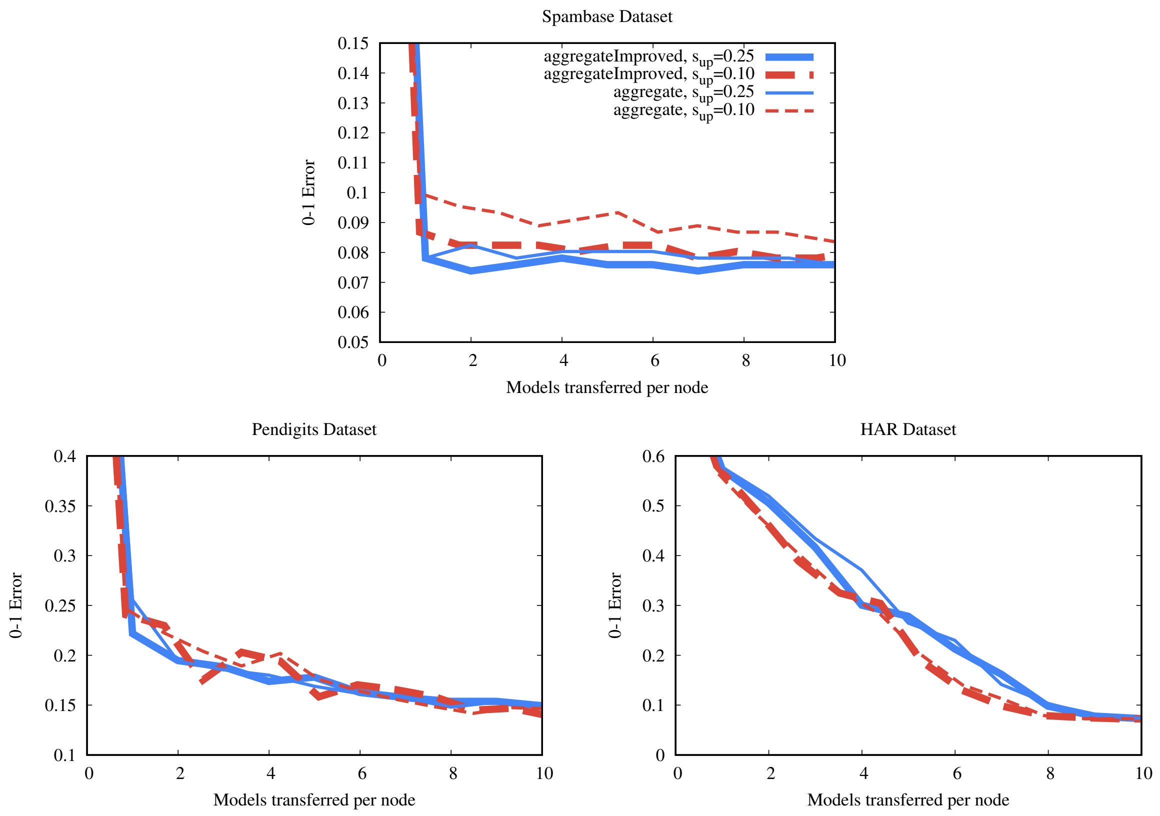

Figure 4.3: *Federated learning, 100 nodes, long transfer time, no failures, different aggregation algorithms and upstream subsampling probabilities and with* $s_{down} = 1$.

for a total of 64 cores. The server had 512 GB RAM. With this configuration, the experiments we include in this chapter can be completed within a month.

We measure learning performance with the help of the 0-1 error, which gives the proportion of the misclassified examples in the test set. In the case of gossip learning, the loss is defined as the average loss over the online nodes. This means that we compute the 0-1 error of all the local models stored at the nodes over the same test set, and we report the average. In federated learning we evaluate the central model at the master. Note that this is often more optimistic than evaluating the average of the online nodes. For example, if the downstream communication is compressed (that is, $s_{down} < 1$), the local models will always be more outdated than the central one, because the nodes will not receive the complete model.

The presented measurements are averages of 5 runs with different random seeds. The only exceptions are the measurements with our gossip algorithms in the scenarios over the HAR dataset when the network size was the database size. These scenarios are costly to simulate so we show a single run.

The 0-1 error is measured as a function of the total amount of bits communicated anywhere in the system normalized by the number of online nodes. We use the size of a full machine learning model as the unit of the transferred information.

4.4.1 Basic Design Choices

First, we compare the two aggregation algorithms for subsampled models in Algorithm 4.6 (Figure 4.3) in the no-failure scenario. The results suggest that AGGREGATEIMPROVED has a slight advantage, although the performance depends on the database. In the following we will apply AGGREGATEIMPROVED as our implementation of the method AGGREGATE.

Another design choice that we study is the subsampling (that is, compression) strategy for federated learning. Recall that we have a choice to subsample only the model that is sent by the client to the master or we can subsample in both directions. Note that if we subsample in both directions, then we can achieve a much higher compression rate, but the convergence will be slower. Overall, however, it is possible that, as a function of the total number of bits communicated, it is still preferable to compress in both directions. Note that subsampling just the model from the master to the client is meaningless because that way the client will send mostly outdated parameters that the master has already received in previous rounds.

Figure 4.4 compares the two meaningful strategies for the case of $s = 0.1$. Subsampling in both directions is clearly the better choice. However, it also has a downside, because in this case the clients no longer receive the full model from the master so they cannot use the best possible model locally. To illustrate this problem, we include the average performance of the models stored locally. Depending on the ap-

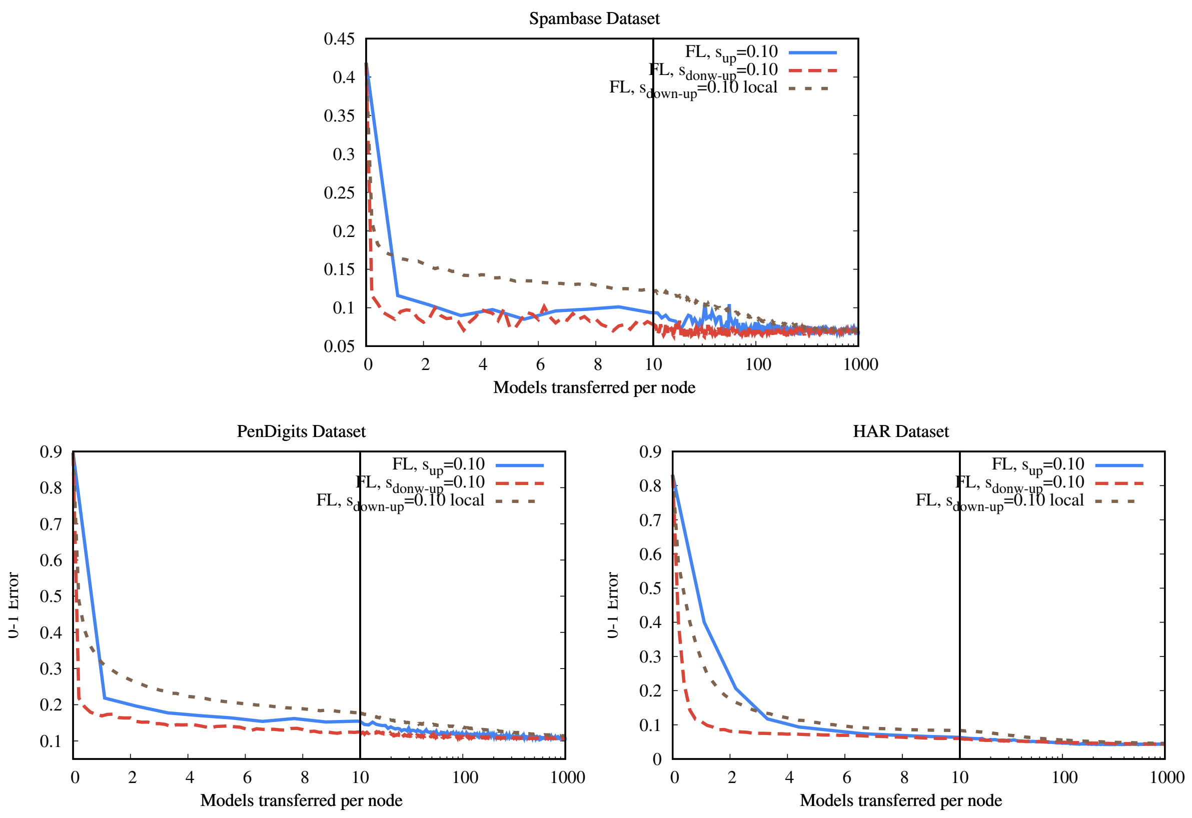

Figure 4.4: *Federated learning with 100 nodes, no-failure scenario, with different sub-sampling strategies and $s = 0.1$. The "local" plot shows the average of the models that the clients store; otherwise the master's model has been evaluated.*

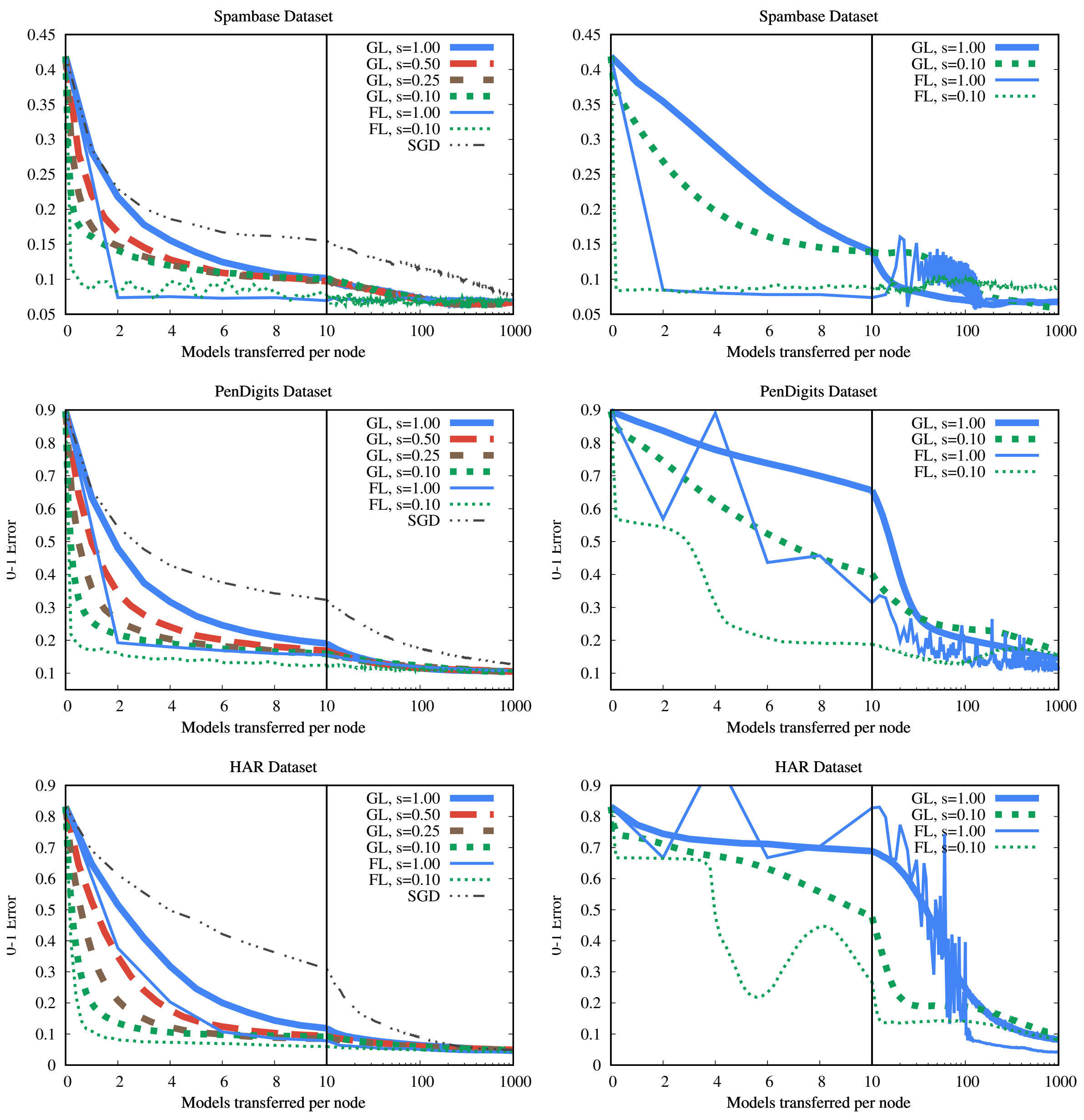

Figure 4.5: *Federated learning and gossip learning with 100 nodes (left) and with one node for each sample (right), no-failure scenario, with different subsampling probabilities. Stochastic Gradient Descent (SGD) is implemented by gossip learning with no merging (received model replaces current model).*

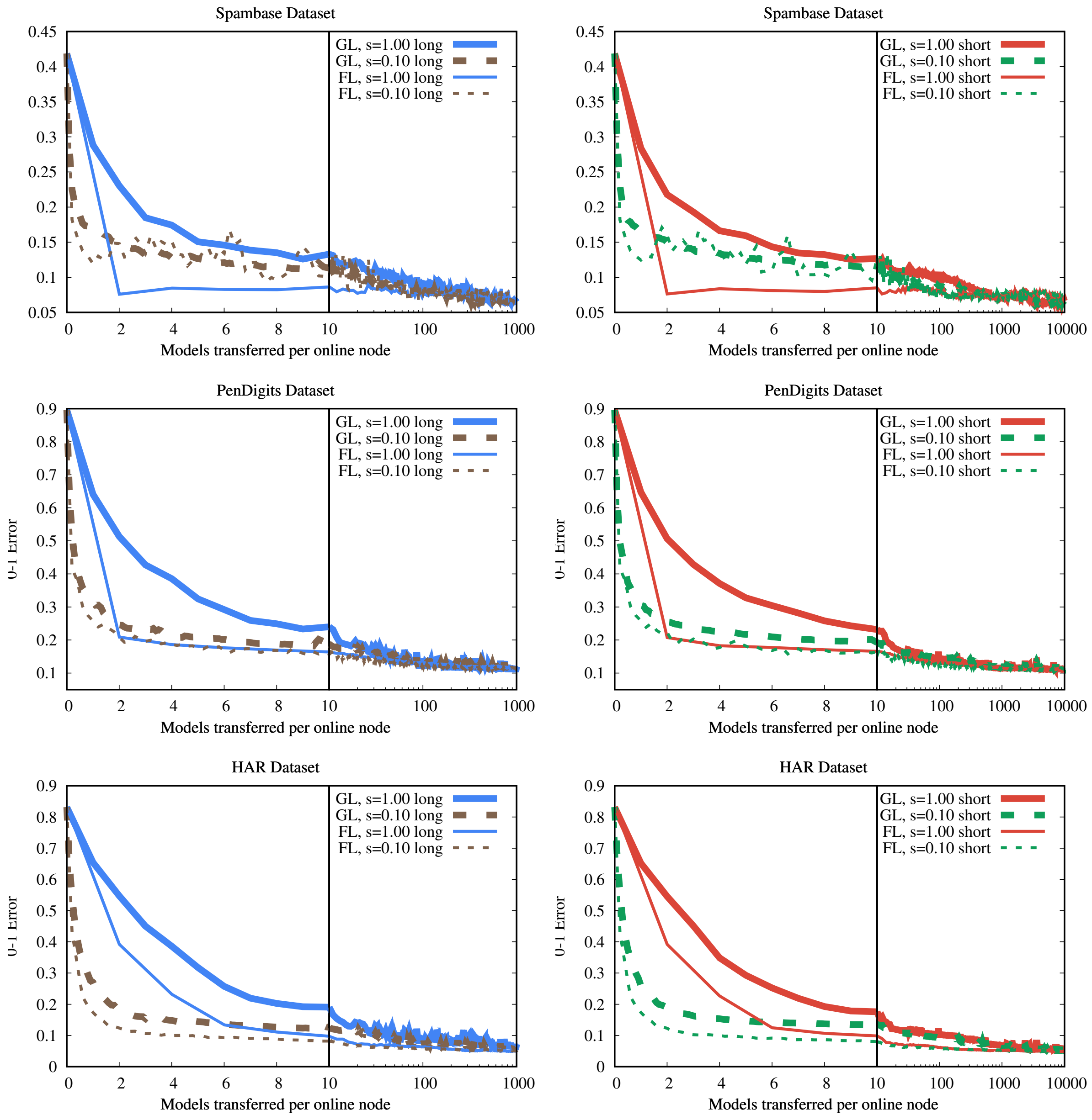

Figure 4.6: *Federated learning and gossip learning over the smartphone trace with long (left) and short (right) transfer time, in the 100-node scenario.*

plication, this may or may not be a problem. Nevertheless, from now on, we will apply subsampling in both directions in the remaining experiments.

4.4.2 Small Scale

Next, we study the case when each example was assigned to a single node. A comparison of the different algorithms and subsampling probabilities is shown in Figure 4.5. The stochastic gradient descent (SGD) method is also shown, which was implemented by gossip learning with no merging, where the received model replaces the current model at the nodes. Clearly, the methods that use merge are all better than SGD. Also, it is quite apparent that subsampling helps both federated learning and gossip learning.

Most importantly, in the 100-node setup (left column of Figure 4.5), gossip learning is competitive with federated learning in the case of high compression rates (that is, low sampling probabilities). This was not expected, as gossip learning is fully decentralized, so the aggregation is clearly delayed compared to federated learning. Indeed, with no compression, federated learning performs better.

Figure 4.5 (right) also shows the extreme scenario, when each node has only one example, and the size of the network equals the dataset size. This is a much more difficult scenario for both gossip and federated learning. Also, federated learning is expected to perform relatively better, because of the more aggressive central aggregation of the relatively little local information. Still, gossip learning is in the same ballpark in terms of performance. In terms of long range convergence (recall that our scenarios cover approximately a time of one day), all the methods achieve good results.

Figure 4.6 contains our results over the smartphone trace churn model. Here, all the experiments shown correspond to a period of 24 hours, so the horizontal axis has a temporal interpretation as well. The choice of a long or short transfer time makes almost no difference (apart from the fact that the shorter transfer time obviously corresponds to a proportionally faster convergence). Also, interestingly, churn only leads to a minor increase in the variance of the 0-1 error but otherwise we have a stable convergence. It is also worth pointing out that federated learning and gossip learning shows a practically identical performance under high compression rates. Again, gossip learning is clearly competitive with federated learning.

Figure 4.7 contains the results of our experiments with the single class assignment scenario, as described in Section 4.3.1. In this extreme scenario, the advantage of federated learning is more apparent, although in the long run gossip learning also achieves good results. Interestingly, in this case the different compression rates do not have any clear preference order. For example, on the Pendigits database (containing 10 classes) the compressed variant is inferior, while on HAR (with 6 classes)

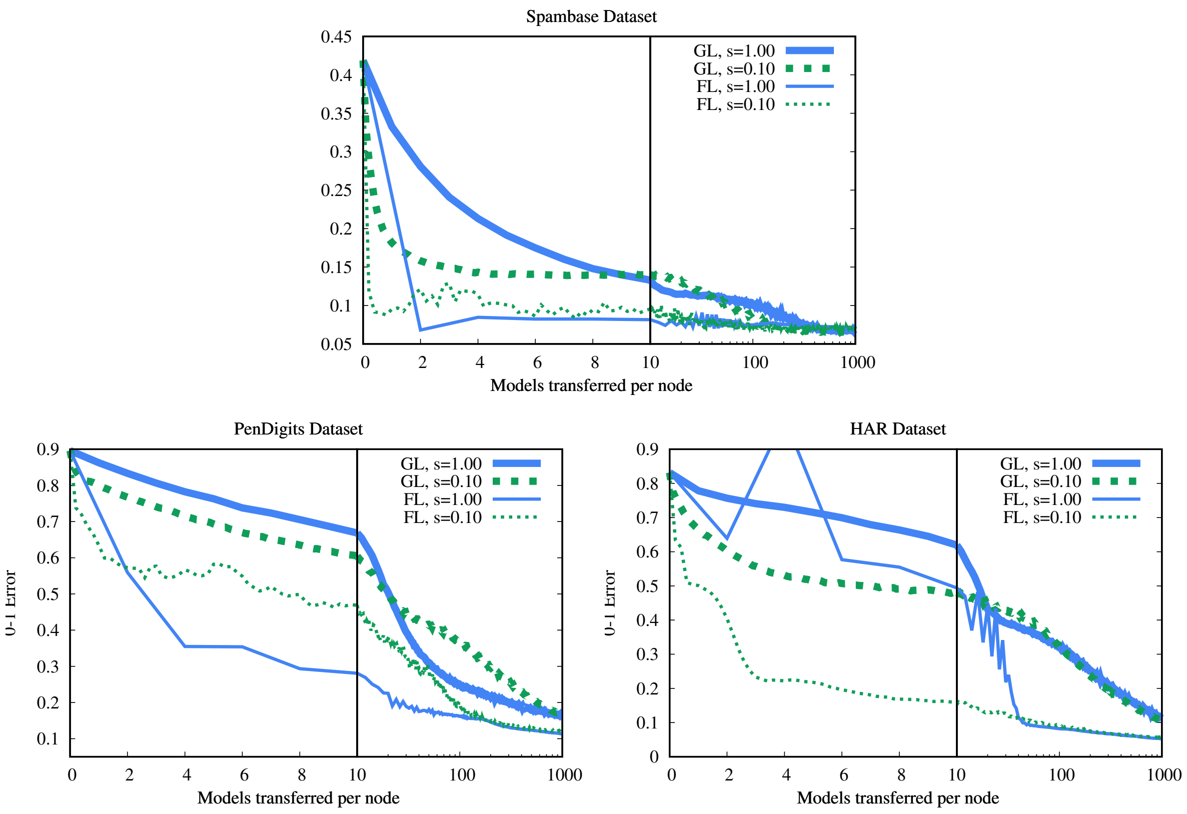

Figure 4.7: *Federated learning and gossip learning with 100 nodes, no-failure scenario, with single class assignment.*

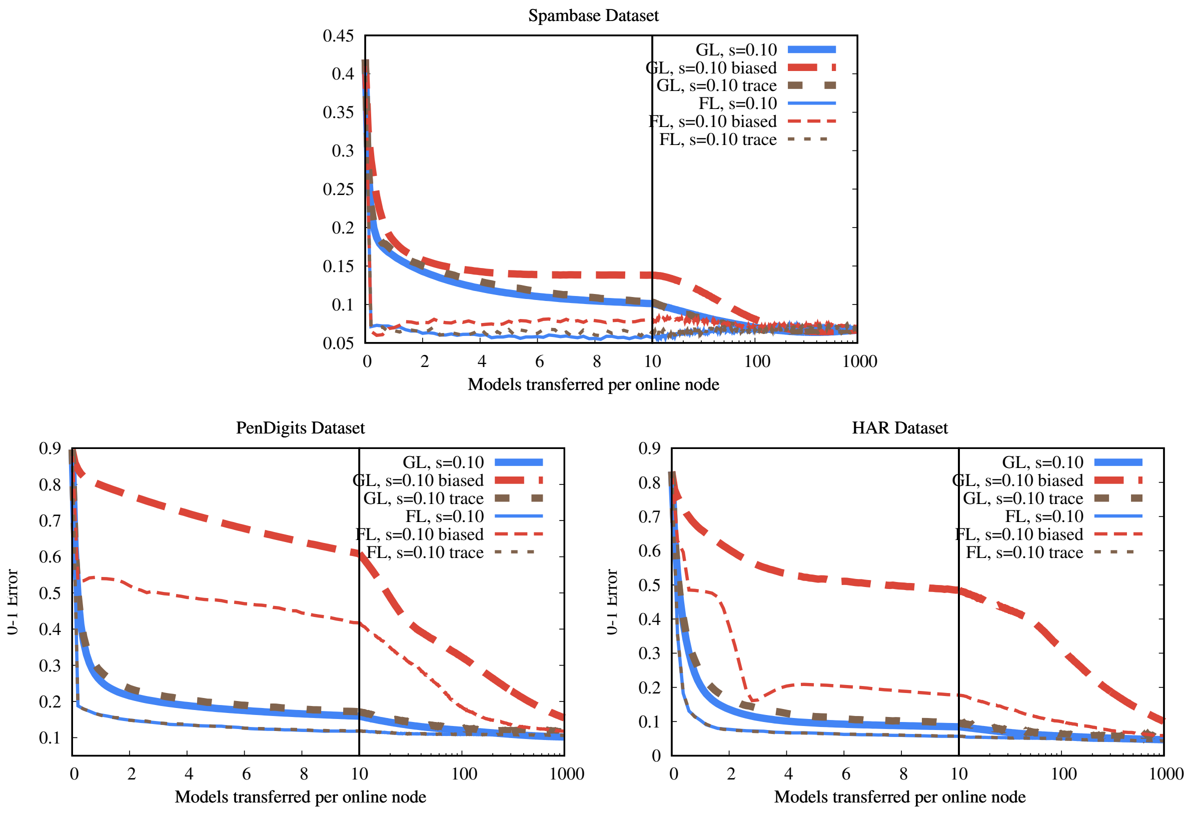

Figure 4.8: *Selected experiments in the large scale scenario. 'Biased' indicates single class assignment, 'trace' indicates the smartphone trace scenario.*

the compressed variant appears to be preferable.

Let us also point out the similarity between the results in Figure 4.7 and Figure 4.5 (right). Indeed, the scenario where each node has one sample is by definition also a single class assignment scenario.

4.4.3 Large Scale

Here, we experiment with the scenario where the number of examples per node was the same as in the 100-node scenario, but the network size equaled the size of the database. To achieve this, we replicated the examples; that is, each example was assigned to multiple nodes (see Section 4.3.1). We call this the large scale scenario.

The results are shown in Figure 4.8. We can see that here the best gossip variants are competitive with the best federated learning variants. The most important feature of this large scale scenario seems to be whether the label distribution is biased (single label assignment) or not (random assignment). The single class assignment (biased) scenario results in a slower convergence for both approaches. However, compared with previous experiments, increasing the size of the network in itself does not slow the protocols down.

4.5 Related Work

The literature on machine learning and, in general, optimization based on decentralized consensus is vast [74]. Our contribution here is a comparison of the efficiency of decentralized and centralized solutions that are based on keeping the data local. Hence, we focus on studies that target the same problem. Savazzi et al. [73] study a number of gossip based decentralized learning methods in the context of industrial IoT applications. They focus on the case where the data distribution is not identical over the nodes. They do not consider compression techniques or other algorithmic enhancements.

Hu et al. [42] introduce a segmentation mechanism similar to ours, but their motivation is different. Their focus is on saturating the bandwidth of all the nodes using P2P connections that have a relatively smaller bandwidth, which means they propose that the nodes should communicate to several peers simultaneously. Sending only a part of the model appears to be beneficial in this scenario. In our case, we focused on convergence speed as a function of overall communication.

Blot et al. [10] compare a number of aggregation schemes for the decentralized aggregation of gradients, including a gossip version based on the weighted push-sum communication scheme [48]. Although the authors do not cite federated learning or gossip learning as such, their theoretical analysis and new algorithm variants are relevant and merit further study.

Lalitha et al. [51] study the scenario where each node can see only a subset of parameters and the task is to learn a Bayesian model collaboratively without a server. The study is mainly theoretical, an experimental evaluation being done with just two nodes, as an illustration.

Jameel et al. [43] focus on the communication topology, and attempt to design an optimal topology that is both fast and communication efficient. They propose a superpeer topology where superpeers form a ring and they all have a number of ordinary peers connected to them.

Lian et al. [54] and Tang et al. [80] introduce gossip algorithms and compare them with the centralized variant. Koloskova et al. [49] improve these algorithms by supporting arbitrary gradient compression. The main contribution in these studies is a theoretical analysis of the synchronized implementation. Their assumptions on the network bandwidth are different from ours. They assume that the server is not unlimited in its bandwidth usage, and they characterize convergence as a function of the number of synchronization epochs. In our study, due to our edge computing motivation, we focused on convergence as a function of system-wide overall communication in various scenarios including realistic node churn. We performed asynchronous measurements along with optimization techniques.

Giaretta and Girdzijauskas [35] present a detailed analysis of the applicability of gossip learning, but without considering federated learning. Their work includes scenarios that we have not discussed here including the effect of topology, and the correlation of communication speed and data distribution.

Ben-Hun and Hoefler [5] very briefly consider gossip alternatives and claim that they have performance issues.

4.6 Conclusions

Here, we compared federated learning and gossip learning to see to what extent doing away with central components—as gossip learning does—harms performance. The first hurdle was designing the experiments. One has to be careful what system model is chosen and what constraints and performance measures are applied. For example, the best algorithm will be very different when we grant a fixed overall communication budget to the system overall but allow for slow execution, or when we give a fixed amount of time and allow for utilizing all the available bandwidth at all the nodes.

Our choice was to allow the nodes to communicate within a configurable bandwidth cap that is uniform over the network, except the master node. Within this model, we were interested in the speed of convergence after a given amount of overall communication.

We observed several interesting phenomena in our various scenarios. In the random class assignment case (when nodes have a random subset of the learning examples) gossip learning is clearly competitive with federated learning. In the single class assignment scenario, federated learning converges faster, since it can mix information more efficiently. This includes the case where every node has only a single example, as it is a special case of single class assignment. However, gossip learning is able to converge as well in a practically realistic time frame. Here, we think that gossip learning could be improved by applying more sophisticated peer sampling methods that are optimized to increase the efficiency of mixing different updates, or by applying a different learning rule, based on momentum methods, for example, Adam [3].

In our experimental setup we opted for putting the same cap on both upstream and downstream traffic in federated learning, as motivated in Section 4.3.2. But even if one removes this assumption and considers downstream traffic completely free, the downstream-compressed federated learning variant will converge only twice as fast, which is a relatively modest difference. Note that in gossip learning there is only peer-to-peer traffic, so there is only one cap.

What is more, we have only examined subsampling as a compression technique.

There are more sophisticated compression techniques available [22] that could potentially be applied in both federated and gossip learning.

Contribution

In this chapter, the contributions of the author were: the partition-based sampling technique; the design and development of churn-related modules of the simulator; participation in the design of the improved aggregation algorithm for federated learning; participation in the planning of experiments; and the optimization of hyperparameters.

Chapter 5

Gossip Learning with Adaptive Flow Control

Token bucket and leaky bucket algorithms and their variants have long been used for traffic shaping in packet switched networks. These algorithms control the rate at which packets are sent from or forwarded by a networked device. The primary motivation for applying such methods is to prevent large bursts of traffic to protect the network and also to enforce quality-of-service contracts by controlling the rate of traffic.

In the application layer, decentralized applications are also confronted by the issue of traffic shaping. However, since applications have many other key characteristics to worry about, such as performance and fault tolerance, traffic shaping methods have not received much emphasis. Take gossip-based broadcast, for example. The conventional approach is to simply adopt a *proactive* design pattern where nodes gossip periodically in regular intervals [26]. This solves the traffic shaping problem (we have a constant rate) so we can focus on other design decisions that are related to performance and fault tolerance.

In this chapter, we challenge this design philosophy. Our main message is that fine details of traffic shaping actually have a profound effect on many key global application characteristics that seem unrelated to traffic shaping at first. For example, as we will show in detail, when gossip-based broadcast is implemented using our token account algorithm instead of the periodic, round-based communication pattern, convergence becomes dramatically faster, approaching the speed of flooding, without sacrificing the rate limiting feature (as flooding does).

The techniques discussed here are applicable in many decentralized asynchronous message passing applications where the main goal is to reach a target global state quickly and cheaply. These applications include gossip-based algorithms, asynchronous (chaotic) numeric algorithms, and distributed data mining as well. The common characteristics of these applications include nodes receiving messages, updating their state based on these messages, and sending messages as a function of their state.

In such applications, there is typically a large degree of freedom regarding the number and the scheduling of the outgoing messages. Unlike in the networking layer, where messages are simply forwarded, here the messages that are received and sent might be decoupled. The current practice does not exploit this design space fully; traffic shaping and its side-effects have not been given enough attention. There are two kinds of popular approaches, namely proactive and reactive. In a proactive approach, each node sends messages periodically, based on the information accumulated in the previous round. The rounds of the nodes in the system may or may not be synchronized. In a reactive approach, nodes immediately send messages whenever their state changes (typically after receiving a message).

In the proactive approach, time is often wasted, since nodes frequently sit on new information, doing nothing until the next round comes. However, traffic shaping is optimal due to the constant rate. In the reactive approach, information is spread much faster initially; however, the amount of traffic and its burstiness is out of control, which might harm the network as well as the application itself. Our goal here is to propose techniques that inherit the best properties of both approaches while avoiding their drawbacks.

We achieve this by generalizing the token bucket algorithm, introducing a family of token account algorithms. In a nutshell, at each node, these algorithms grant one token to the node in regular periods, and spend a token when the node sends a message. The details of when to send messages, how many, and how exactly to limit the number of accumulated tokens are captured with two functions: the proactive and reactive functions. This design space includes the purely proactive and reactive protocols and a spectrum of algorithms in between.

It should be mentioned that, unlike token bucket algorithms, our token account protocols are targeted to serve the application layer where they fulfill many functions at once, all of which are equally important: *rate limiting*, *speeding up convergence* and *fault tolerance*. The speedup effect is due to the reactive behavior that reduces the idle time, during which nodes sit on new information. Fault tolerance is due to the proactive behavior that maintains a certain level of messaging activity even when messages are lost due to faults or due to the semantics of the application.

Our contribution here is twofold. First, we introduce the token account service along with three different implementations. The parameters of these implementations allow us to span the design space between proactive and reactive algorithms. We evaluate the proposed token account protocols using three applications (gossip learning, push gossip and chaotic power iteration) in simulation. Second, we adapt this framework to partitioned gossip learning, and evalute it in multiple machine learning scenarios, using federated learning as a baseline.

Algorithm 5.1 Push gossip

```
1: update ← null
2: loop
3:     wait($\Delta$)
4:     $p$ ← selectPeer()
5:     send update to $p$
6: end loop
7: procedure ONUPDATE($m$)
8:     if $m$ is fresher than update then
9:         update ← $m$
10:    end if
11: end procedure
```

5.1 Background

For our system model, see Section 2.2. Here, we describe the three applications we selected to test our token account service: gossip learning, gossip-based broadcasting and chaotic power iteration. These applications are all based on local message passing and their goal is to converge to a desirable state through an iterative process as quickly and as cheaply as possible. Yet, they have a rather diverse set of requirements that allow us to demonstrate the broad applicability of our algorithms and to better cover their advantages and limitations. We first present the most common, proactive implementation of the demonstrator applications, and later on we shall reformulate them over the token account API (Algorithm 5.3).

5.1.1 Gossip Learning

Our first demonstrator application that can take advantage of our token account service is gossip learning [65]. The basic idea is that in the network many models perform random walks and are updated at every node using the local example. For now, we will use the version described in Section 2.3, but instead of performing actual model merging, we pick the better-trained model (in terms of model age). While model merging is often beneficial, it may be highly non-trivial in the case of non-parametric machine learning models. Furthermore, abstaining from merging enables us to evaluate the token account framework independently from specific machine learning tasks, since the model performance will approximately be a function of model age. Later, in Section 5.2.3 we will use model merging with a new token account variant.

5.1.2 Push Gossip

Our second example application is the classical push gossip protocol [26], as shown in Algorithm 5.1.

In this simple setup, we assume that every node stores a single update, and whenever a new, fresher update arrives, it replaces the old one. Furthermore, all the nodes periodically push the update they know about to a neighboring node. Here, we do not consider any stopping criteria as we assume that updates arrive frequently and continuously.

Although the push-pull variant is superior to push according to a number of performance metrics, and it could also be used alongside our token account service, we chose push for the sake of simplicity. This is because pull variants have benefits mainly in the final phase of convergence, which (as confirmed by our preliminary experiments) is not actually observed in our setup here due to the continuous stream of new updates.

5.1.3 Chaotic Asynchronous Power Iteration

Our third example is power iteration. Given a square matrix A, vector x is an *eigenvector* of A with *eigenvalue* λ, if $Ax = \lambda x$. Vector x is a *dominant* eigenvector if there are no other eigenvectors with an eigenvalue larger than $|\lambda|$ in absolute value. In this case λ is a *dominant eigenvalue* and $|\lambda|$ is the *spectral radius* of A.

We concentrate of the abstract problem of calculating the dominant eigenvector of a weighted neighborhood matrix of some large network, in a decentralized way, when the elements of the vector are held by individual network nodes, one vector element per node. The matrix A is defined by physical or overlay *links* between the network nodes. More precisely, A contains the *weights* assigned to these links: let matrix element A_{ij} be the weight of the link from node j to node i. If there is no link from j to i then $A_{ij} = 0$.

In [56], Lubachevsky and Mitra present a chaotic asynchronous family of message passing algorithms to calculate the dominant eigenvector of a non-negative irreducible matrix, that has a spectral radius of one. Algorithm 5.2 shows an instantiation of this framework, that we will apply here.

In the algorithm, the values x_i represent the elements of the vector that converge to the dominant eigenvector. The values b_{ki} are buffered incoming weighted values from incoming neighbors in the graph. These values are not necessarily up-to-date, however, as shown in [56], the only assumption about message failure is that there is a finite upper bound on the age of these values. The age of value b_{ki} is defined by the time that elapsed since k sent the last update successfully received by i. This bound can be very large, so delays and message drop are tolerated to a very large extent.

Algorithm 5.2 Asynchronous iteration executed at node i

```
1: $b_{ki} \leftarrow$ any positive value for all k
2: loop
3:     wait($\Delta$)
4:     $x_i \leftarrow \sum_{k \in \text{in-neighbors}_i} A_{ik} b_{ki}$
5:     $p \leftarrow$ selectPeer()
6:     send weight $x_i$ to $p$
7: end loop
8: procedure ONWEIGHT($m$)
9:     $k \leftarrow$ m.sender
10:    $b_{ki} \leftarrow m.x$
11: end procedure
```

5.2 Token Account Algorithms

The example algorithms presented so far were fully proactive, sending messages in regular time intervals. This provides excellent load balancing, but slows down convergence. We could consider the naive reactive variants of these algorithms, where, instead of a regular timer, every message received would trigger message sending immediately. This would result in a faster convergence but the uncontrolled communication load would lead to large bursts of traffic. In our framework we introduce an abstraction that allows for a fine control over the tradeoff between these two approaches.

One idea to achieve this tradeoff is to apply the token bucket algorithm. In this algorithm, a token is assigned to the node in regular intervals of length Δ. The application works in purely reactive mode, spending one token per message. If no tokens are available, no sending is allowed (so sending is either skipped or blocked, depending on the application semantics). Our approach is similar in spirit, but it offers a fine control over the proactive and reactive characteristics of the application and it also allows for application specific adaptation in a natural manner. This allows us to achieve almost optimal speedup while preventing uncontrolled flooding and providing fault tolerance as well.

5.2.1 Token Account Framework

In our framework, each node has an *account,* which can hold a non-negative integer number of tokens. We introduce two functions that will control the proactive and reactive behavior of the node as a function of the number of tokens.

The *proactive function* PROACTIVE(a) returns the probability of sending a proactive message as a function of the account balance a. We require that the proactive function should be monotone non-decreasing in a, that is, a higher balance should not

Algorithm 5.3 Token account

```
 1: $a \leftarrow$ initial number of tokens
 2: loop
 3:     wait($\Delta$)
 4:     do with probability proactive($a$)
 5:         $p \leftarrow$ selectPeer()
 6:         $m \leftarrow$ createMessage()
 7:         send $m$ to $p$
 8:     else
 9:         $a \leftarrow a + 1$
10:     end do
11: end loop
12: procedure ONMESSAGE($m$)
13:     $u \leftarrow$ updateState($m$)
14:     $x \leftarrow$ randRound(reactive($a, u$))
15:     $a \leftarrow a - x$
16:     for $i \leftarrow 1$ to $x$ do
17:         $p \leftarrow$ selectPeer()
18:         $m \leftarrow$ createMessage()
19:         send $m$ to $p$
20:     end for
21: end procedure
```

result in a lower probability of sending a proactive message.

The *reactive function* REACTIVE(a, u) returns the number of messages that the node will send as a reaction to an incoming message, as a function of the account balance a and the usefulness of the received message u. Clearly, the higher the balance the more messages we might want to send so the function should be monotone non-decreasing in a. The usefulness u expresses the notion that some messages are more important than others in most applications. For example, in the broadcast application, the received message is useful if and only if it contains new information for the node. Currently we assume that u is either 1 or 0 (the message is either useful or not). Finer grading is possible in the future. The function should be monotone non-decreasing in u as well, that is, more useful messages should not result in fewer reactive messages being sent. Also, the value returned is at most a (we do not allow overspending).

The purely proactive strategy is a special case given by PROACTIVE$(a) \equiv 1$ and REACTIVE$(a, u) \equiv 0$. With relaxing the non-negativity constraint of the balance, the purely reactive strategy can be expressed as well as PROACTIVE$(a) \equiv 0$ and REACTIVE(a, u) $\equiv k$ (or REACTIVE$(a, u) \equiv uk$) for a constant $k \geq 1$.

The pseudo-code for the token account algorithm is shown in Algorithm 5.3. In each round, the node either sends a message to a peer, or saves the token for later use; the former occurs with probability PROACTIVE(a). When receiving a message, the application-specific code updates the state of the node using method UPDATESTATE() that also returns the usefulness of the received message. Next, the reactive function returns the number of messages to be sent and the same number of tokens are removed from the account. The return value r of the reactive function is probabilistically rounded by sampling $\lfloor r \rfloor + \xi$ where ξ is a random variable with the distribution $\xi \sim \text{Bernoulli}(r - \lfloor r \rfloor)$.

The framework can be instantiated by implementing the proactive and reactive functions. We will discuss our proposed implementations in Section 5.2.4. First, however, we turn to the implementation of our three application examples within the framework.

5.2.2 Applications within the Framework

To implement our applications in the framework we have to provide the application specific implementations of two methods: CREATEMESSAGE() that is responsible for constructing a message to be sent based on the current state, and UPDATESTATE(m) that is responsible for updating the current state based on the new message that has been received. This includes defining the usefulness of the received message m because UPDATESTATE(m) has to return this information.

The implementation of CREATEMESSAGE() is simple in all three cases: we just copy the current state. In the gossip learning application, the state consists of a machine learning model with a counter (age) that keeps track of how many times the given model was updated. In the push gossip application the state consist of an update with a timestamp. In chaotic iteration the state is the value x_i at node i.

In our three applications, the implementations of UPDATESTATE(m) are given by ONMODEL, ONUPDATE and ONWEIGHT, respectively. In addition, we have to return usefulness, as we explain below. In gossip learning, usefulness is 0 if the current model of the node is older (in terms of the number of visited nodes) than the received model, and 1 otherwise. In the former case, the state is unchanged, while in the latter case, the received model is trained on the local data and stored as the new state. Note that in our simulations, regarding this basic version of gossip learning, we did not implement any actual machine learning tasks, but just simulated the age of the models as this forms the basis of our performance metric.

In the broadcast application, usefulness is 1 if and only if the received message contains a newer update than the locally stored update at the node. In our simulations, we considered the following scenario: new updates are regularly injected into random online nodes of the network. A newer update makes older updates obsolete, that is, only the freshest update known by the given node is stored and propagated. Our performance metric is the difference between the average times-

Algorithm 5.4 Partitioned Token Gossip Learning

```
1: $(t, w, b) \leftarrow (\mathbf{0}, \mathbf{0}, 0)$
2: $a \leftarrow \mathbf{0}$
3: loop
4:     wait($\Delta_g$)
5:     $j \leftarrow$ selectPart()                          ▷ select a random partition
6:     do with probability proactive($a[j]$)
7:         $p \leftarrow$ selectPeer()
8:         send samplePartition($t, w, b, j$) to $p$
9:     else
10:        $a[j] \leftarrow a[j] + 1$          ▷ we did not spend the token so it accumulates
11:    end do
12: end loop
13:
14: procedure onReceiveModel($t_r, w_r, b_r, j$)
15:    $(t, w, b) \leftarrow$ merge($(t, w, b), (t_r, w_r, b_r)$)
16:    $(t, w, b) \leftarrow$ update($(t, w, b), D$)
17:    $x \leftarrow$ randRound(reactive($a[j]$,1))
18:    $a[j] \leftarrow a[j] - x$                                  ▷ we spend $x$ tokens
19:    for $i \leftarrow 1$ to $x$ do
20:        $p \leftarrow$ selectPeer()
21:        send samplePartition($t, w, b, j$) to $p$
22:    end for
23: end procedure
```

tamp of the freshest update known by each node and the timestamp of the freshest update in the whole network.

In the chaotic iteration application, usefulness is 1 if and only if the received message causes a change in the local state. Our convergence metric is the angle (or cosine distance) between the approximation of the eigenvector and the actual eigenvector that should converge to zero.

5.2.3 Partitioned Token Gossip Learning

In Section 4.1, we introduced gossip learning with sampling. Using the partitioned sampling, merge and update methods described there, we propose partitioned token gossip learning, shown in Algorithm 5.4.

The token account algorithm allows chains of messages to form that travel fast in the network like a "hot potato". Therefore, it is vital that we use samplePartition as an implementation of sampling. Our preliminary experiments showed that random sampling (sampleRand) is not effective along with the token account technique. This is because if we sample independently at each hop, we work strongly against the formation of long “hot potato” message chains that represent any fixed model parameter. This is a key insight, and this is why the partitioned approach is expected to work better. It allows for hot potato message chains to form based on a single partition. In other words, we can have the benefits of sampling-based compression and hot potato message passing at the same time.

Accordingly, each partition j now has its own token account $a[j]$ that stores the number of available tokens. With this modification of the token account framework, each partition will perform random walks independently, using its own communication budget. Of course, the model update is not independent; it is the same as in partitioned gossip learning. We also track the age of each partition, as discussed previously in connection with the merge and update functions. We will argue that even the proactive gossip learning can benefit from using partitions, although to a lesser extent.

Since we apply merging, the state is always changed, hence we use $u = 1$ on line 17.

If the number of partitions is $S = 1$, then we get a special case of the algorithm without any compression (that is, no sampling), namely we always send the entire model.

Also, let us mention that the methods SELECTPART and SELECTPEER were implemented using sampling without replacement; when the pool of available options becomes empty, it is re-initialized. This is slightly better than sampling with replacement, because it results in a lower variance.

5.2.4 Implementations of the Framework

Let us turn to the instantiations of the framework. In order to implement the framework, one has to provide the two functions PROACTIVE(a) and REACTIVE(a,u) taking into account the constraints we described previously. We have already described the implementation of the purely proactive solution within the framework as an example. Here we propose three additional implementations.

Simple token account

The first implementation is called *simple* token account. This implementation serves as a baseline, and it is similar to the token bucket algorithm although there are important differences as well. We introduce a parameter $C \geq 0$ that controls the capacity of the token account. That is, the maximal number of tokens will be C. Using this parameter, we define

$$\text{PROACTIVE}(a) = \begin{cases} 1 & \text{if } a \geq C \\ 0 & \text{otherwise}, \end{cases} \tag{5.1}$$

$$\text{REACTIVE}(a,u) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

Note that when $C = 0$ we have the purely proactive protocol as a special case. The reactive part is identical to that of the token bucket algorithm, however, this implementation also shows proactive behavior but only when the account is full. The account typically fills up with tokens when too few messages are arriving relative to the allowed communication rate. This in turn happens most often when, due to failures, fewer and fewer messages are circulating in the network. The default proactive behavior helps maintain a certain level of communication rate naturally even under high message drop rates, which is impossible in a purely reactive implementation. Of course, the effect of this error correction strongly depends on the application semantics.

Generalized token account

Our second implementation is called *generalized* token account. Here, our goal is to design a reactive function that is able to increase the number of messages sent when the number of tokens is high. In addition, we want to send twice as many messages in response to useful messages. To achieve this goal, the proactive function should be the same as the one in (5.1) and we propose the following reactive function:

$$\text{REACTIVE}(a,u) = \begin{cases} \lfloor (A-1+a)/A \rfloor & \text{if } u = 1 \\ \lfloor (A-1+a)/(2A) \rfloor & \text{otherwise,} \end{cases} \tag{5.3}$$

where parameter A is a positive integer and it controls what proportion of the available tokens we wish to use. Let us first consider the case when the incoming message was useful ($u = 1$). Here, the reactive function is designed so that when $A = 1$ we use all the available tokens. Increasing A decreases the returned value. When $A \geq a$, the function returns 1. This also implies that the maximal meaningful value for A is $A = C$ in which case the reactive function will be equivalent to equation (5.2). Now, let us consider the case when $u = 0$. Here, we simply divide the returned value by two. This also means that, due to rounding the output down to an integer, the function will return 0 when $A \geq a$. In other words, when the tokens are scarce, we do not waste them for reacting to messages that are not useful.

Randomized token account

So far all the strategies had the simple proactive function in equation (5.1). In the *randomized* token account implementation we propose a more fine-grained handling of proactive messages, while we will treat reactive messages in a similar way to the generalized token account implementation. In addition, instead of rounding it down to an integer, the reactive function will use the value of a similar formula as the expected value of a discrete distribution, from which a sample is returned.

Let us first discuss the proactive function given by

$$\textsc{proactive}(a) = \begin{cases} 0 & \text{if } a < A-1 \\ \dfrac{a-A+1}{C-A+1} & \text{if } a \in [A-1, C] \\ 1 & \text{otherwise}. \end{cases} \tag{5.4}$$

Parameters A and C have the same semantics as in previous implementations: C controls the capacity of the account, A controls the rate of spending the tokens. The actual formula might seem slightly ad-hoc, but it is derived from a few simple requirements. First, we wish the function to return 1 when $a \geq C$ as in all previous implementations. Second, we wish to add some proactive behavior even when $a < C$, so the returned value was chosen to be linear starting from $a = A - 1$ until $a = C$. The starting point of this linear segment was chosen to be $A - 1$ because if $a < A$ then the reactive function (to be discussed below) will be able to send less than one messages on average (in other words, we are not guaranteed to be able to respond to important messages) so in that range we wish to maintain the purely reactive behavior.

The reactive function is given by

$$\textsc{reactive}(a, u) = \begin{cases} a/A & \text{if } u = 1 \\ 0 & \text{otherwise}. \end{cases} \tag{5.5}$$

Note that this time we apply no rounding, so the returned value might be less than 1. As shown in Algorithm 5.3, a randomized rounding is performed on this value to get an integer.

5.2.5 A Note on Rate Limitation Properties

The algorithm variants above have rather different reactive functions, some of them allowing for spending the full account at once. This means that the largest possible burst of traffic is defined by the largest possible account balance. Let us take a closer look at the maximum possible size of the account balance. For an arbitrary implementation of the token balance framework, let C be the smallest number for which $\textsc{proactive}(C) = 1$ holds. If there is no such C, it means the balance of the account might in principle grow indefinitely, which is not a desirable property, since we wish to limit the size of bursts. In our implementations we have such a C, in fact, it is an explicit parameter. Due to the definition of C, any additional tokens are guaranteed to be spent immediately when the account has at least C tokens. We call C the *token capacity* of the token strategy, that is, the maximal number of tokens that can be accumulated. This also gives an upper bound on the number of messages that a node may send within a period of time t: a node cannot send more than $\lceil t/\Delta \rceil + C$ messages, where Δ is the length of a proactive round.

5.3 Experimental Analysis of Token Account

The overall goal of our experiments is to examine the speedup of our token account solutions relative to the baseline proactive implementations, while keeping the same overall communication rate. In order to evaluate our protocols, we ran simulation experiments using the PeerSim simulation environment [59]. The evaluation of partitioned token gossip learning merits its own set of experiments, discussed in Section 5.4.

5.3.1 Experimental Setup

The protocols we test consist of the combination of our three applications (gossip learning, push gossip, and chaotic iteration) and our three proposed instantiations of the token account framework: simple token account, generalized token account and randomized token account. These applications and implementations are described in sections 5.1 and 5.2 in detail. The token account protocols have two parameters: A and C. In our experiments we explore this parameter space.

The baseline proactive protocol is given as a special case of simple token account with $C = 0$; this variant is also included in our experiments. Note that the other extreme of the spectrum, namely the pure reactive strategy, is not included as a baseline, since it is obviously not a viable strategy. Without any rate control, our applications would generate a continuous burst and use up all the available bandwidth.

The number of initial tokens assigned to the nodes before the start of the experiment is zero. The communication topology (that is, the overlay network) was a fixed 20-out network (each node had 20 out neighbors that did not change through the experiment) and the network size was $N = 5,000$ or $N = 500,000$. The fixed 20 neighbors were drawn independently and uniformly at random. This is perhaps the simplest practical approximation of uniform peer sampling suitable for the applications we study here. It can be implemented by maintaining 20 TCP connections for the lifetime of the application. The value 20 allows for a robust connected network while the cost of managing the connections to all the 20 neighbors is still practically affordable. The chaotic iteration experiment uses a different topology as we describe later; this is because the 20-out network mixes too well and power iteration converges too fast over this topology.

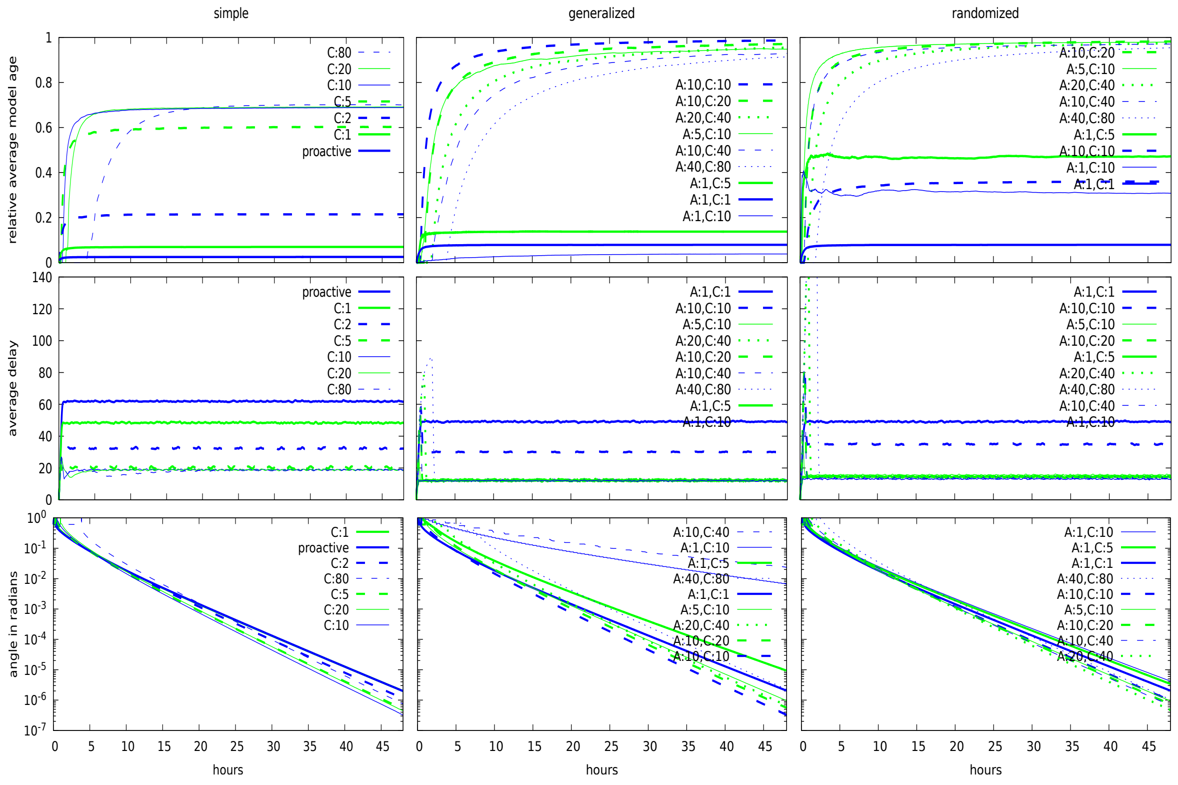

Figure 5.1: *Token account strategies in the failure-free scenario for gossip learning (top row), push gossip (middle row) and chaotic iteration (bottom row).*

As for timing, we simulate a virtual two-day period, with $\Delta = 172.80$ s, allowing for 1000 periods during the two-day interval. This is a long period so we allow only a very low utilization of the available bandwidth in all the applications, which is consistent with the requirements in the domains we target. In all the applications, we assume the transfer time for one message to be 1.728 s, a hundredth of the proactive period. Again, the point here is that we wish to simulate a scenario where low bandwidth utilization is required, because in such a scenario it is much harder to achieve a convergence speed competitive with that of the purely reactive solution that utilizes all the bandwidth.

Regarding the failure patterns, we simulate the protocols in two scenarios. In the first scenario the communication and the nodes are reliable. In the second scenario, we simulate the protocols over a smartphone trace that captures realistic failure and accessibility patterns. In both cases, the same random 20-out network is used as the communication overlay, as described above. We used the trace described in Section 2.4.

Let us now describe the specific settings for each application.

Gossip learning setup

Our goal is to study the speed of convergence. In the case of gossip learning, the learning speed depends on how many nodes a given machine learning model can visit in a given amount of time. (We assume that each node in the network has only one training example.) The maximal number of visited nodes at time t (let us denote this by $n^*(t)$) is achieved by the pure reactive strategy, where no model is ever delayed at any of the nodes. Since the transfer time for one model was assumed to be 1.728 s, at any point in time t we have $n^*(t) = t/1.728$. Our performance metric is defined as the relative number of visited nodes compared to this ideal number. More precisely, let $n_i(t)(\leq n^*(t))$ denote the number of nodes that the model at node i has visited up to time t. Our performance metric at time t is

$$\frac{1}{N}\sum_{i=1}^{N}\frac{n_i(t)}{n^*(t)} = \frac{1}{Nn^*(t)}\sum_{i=1}^{N}n_i(t), \tag{5.6}$$

where N is the size of the network. This describes the relative speed of our protocols compared to the maximal speed. Note that no actual machine learning task is necessary for this metric.

Push gossip setup

In the case of push gossip, the spreading of a single update is relatively difficult to evaluate. For this reason, we inject new updates in regular time intervals at randomly selected nodes in the network. The period of inserting new updates is 17.28 s, that is, we insert 10 updates in every proactive period. Updates have a timestamp so every node can replace a locally stored update by a newer one. Our performance metric at time t is based on the average time lag experienced by the nodes relative to the freshest update available anywhere in the network at time t. For the sake of simplifying our notation, let us assume that at time t the freshest update is the t-th update, and let node i store the t_i-th update ($t_i \leq t$). Our performance metric for push gossip is

$$\frac{1}{N}\sum_{i=1}^{N}(t - t_i) = t - \frac{1}{N}\sum_{i=1}^{N}t_i. \tag{5.7}$$

In the churn scenario, nodes that come back online first send a single initial pull request to a random online neighbor. If this neighbor has tokens, a message is sent back with the latest update (burning a token). Otherwise, no answer is given so the pull request is unsuccessful.

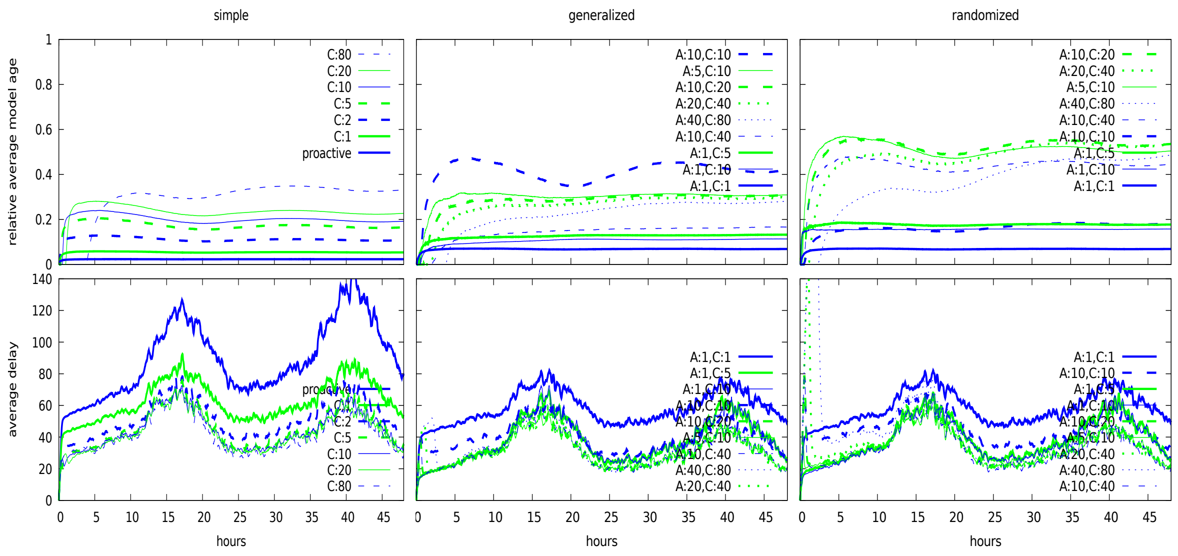

Figure 5.2: *Token account strategies in the smartphone trace scenario for gossip learning (top row) and push gossip (bottom row).*

Chaotic iteration setup

In this application the overlay network not only defines the communication channels but it defines the computational task as well, since we are calculating the eigenvector of the normalized adjacency matrix itself. The 20-out matrix used in the other applications is not suitable because it converges very fast due to the good mixing properties of the network, which hides the effects of the different protocols. Here, we use an overlay network based on the Watts-Strogatz model in order to be able to control (slow down) the speed of convergence [83]. The network is based on a ring in which every node is connected to its closest 4 neighbors. In addition, we rewire every link to a random target with a probability of 0.01. The network size remains $N = 5000$.

The performance metric used in this application is simply the convergence rate of power iteration to the correct eigenvector expressed as the angle of the current approximation and the correct eigenvector. An angle of zero means a perfect solution. In the case of power iteration, there is no natural optimally fast protocol since the convergence speed also depends on the choice of the neighbors. Here, we simply present the convergence as a function of time for the different parameter settings, which still allows for a clear comparison among the different options.

5.3.2 Experimental Results

We first explored the parameter space of the protocols. The parameter space included all the combinations defined by $A = 1, 2, 5, 10, 15, 20, 40$ and $C - A = 0, 1, 2, 5, 10, 15, 20, 40, 80$ (note that we have to have $A \leq C$). Based on these runs a representative selection is shown in Figure 5.1 for our three applications in the failure-free scenario. We performed 10 independent runs for every parameter combination, and the average of these runs is shown in the plots. On the plots showing push gossip we applied smoothing based on averaging measurements over 15 minute periods.

Note that in general it makes little sense to set C much larger than A, since a small A means an aggressive reactive message strategy (we spend most of our tokens), while a large C represents a very low probability of sending proactive messages. This combination results in a very poor error correction ability: if the number of messages in circulation decreases due to faults or due to the application semantics, we cannot replace them efficiently with proactive messages. This is because the aggressive reactive strategy quickly uses up all the tokens, but it takes a very long time until the account is full again (and so proactive messages can be sent).

The main conclusion from this exploration is that, relative to our purely proactive baseline, all the parameter combinations result in a very significant performance improvement in the case of gossip learning and push gossip, and we can also improve chaotic iteration significantly with most parameter combinations.

In the case of push gossip most of the parameter settings result in an almost identical performance, except two settings that are inferior. Clearly, in the broadcast example, it makes sense to be more aggressive in the reactive function and spread the fresh information to multiple nodes when possible; with $A = C$, only at most one reactive message is sent. Gossip learning is more sensitive to the parameter setting. Here, the key appears to be setting a large enough C, which allows us to accommodate the maximal variance in the number of random walks forwarded within a round. Fortunately, settings as low as $C = 20$ already provide close to optimal performance while still offering good rate limiting as well. Note also that larger values of C have a handicap in our experiments since we initialize the accounts to have zero tokens. In the long run, this disadvantage disappears.

It is interesting to note that some settings, such as $A = 10, C = 10$, behave quite differently in different applications. This setting is among the worst in push gossip for reasons mentioned above but it is the best in gossip learning and chaotic iteration. At the same time, $A = 10, C = 20$ is among the best in all three applications.

For the gossip learning application the results have an interesting implication. In this case, machine learning models walk nearly without any delay but the overall communication in the system is not more than in the proactive case. This is possible only if the number of models that walk in the network is less than that in the proactive case. In other words, the token account service has a side-effect of reducing the

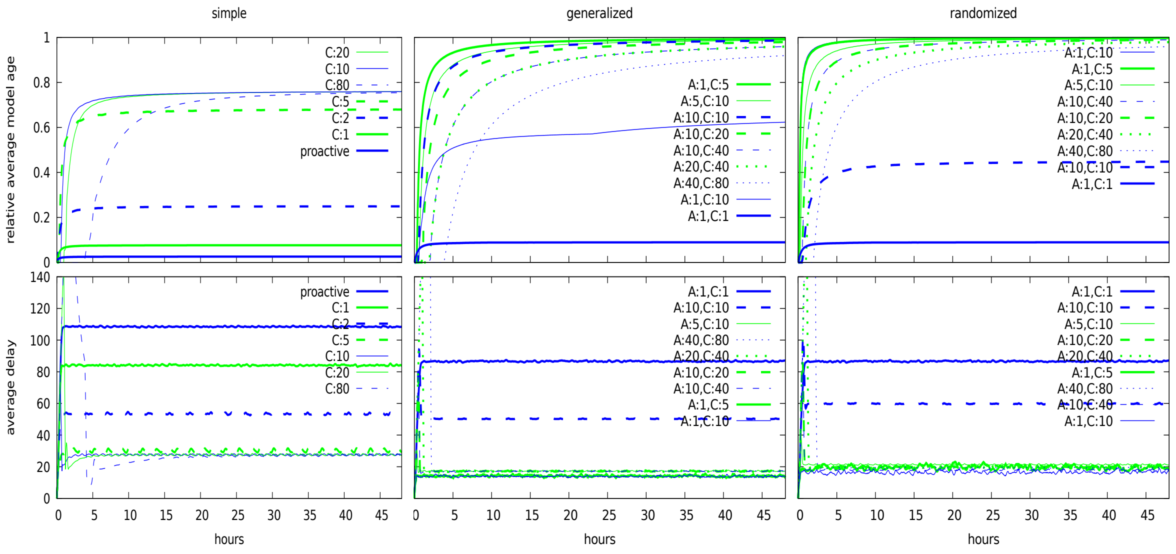

Figure 5.3: *Token account strategies in the failure free scenario and* $N = 500,000$ *for gossip learning (top row) and push gossip (bottom row).*

number of models at the cost of speeding them up at the same time. In fact, we can observe an emergent evolutionary process in which random walks fight for bandwidth and only those survive that happen to reach a given node the soonest after the node received a token.

We performed the same exploration over the smartphone trace as well. Figure 5.2 illustrates the same parameter combinations as shown in the failure-free case. Note that nodes only receive tokens when online (and thus have a chance of actually spending it) and only the online nodes were considered when computing our performance metrics. The chaotic iteration application is not shown here, because in such an extremely dynamic setting with aggressive churn it is not possible to define convergence for this application and so our performance metric is not applicable. Apart from the apparent diurnal pattern due to the variation of node availability, the results are rather consistent with those in the failure-free scenario. Relative to the proactive strategy we achieve very significant improvements, of course, with the same overall communication cost as in the proactive strategy.

To illustrate the scalability of the protocols, we ran them over a network of size $N = 500,000$ in the failure free scenario. The results are shown in Figure 5.3. Comparing with the plots in Figure 5.1, it is clear that in the case of push gossip the protocols are still very robust to the parameter settings, since all the settings that allow for an exponential spreading of new updates (that is, where $C > A$) still have an almost identical performance. Of course, the average delay increases somewhat, but this is due to the larger diameter of the network: a logarithmic increase is expected even with flooding (the reactive variant) with increasing network size (note that our overlay network has a logarithmic diameter).

In the case of gossip learning, we can see that some of the best variants perform very similarly over different network sizes, with two notable exceptions: $A = 1, C = 5$, and $A = 1, C = 10$. These variants were among the worst in the small network but they are among the best in the large network. Note that these variants are the most aggressive reactive variants, they replicate the good random walks burning all the available tokens locally. The reason for the dramatic difference is that—due to finite size effects—in the small network all the random walks get stalled periodically, effectively rendering the dynamics similar to that of the proactive protocol. In the large network there are proportionally more random walks and at every point in time a few of these walks can still make progress and later also replicate to replace those walks that were less lucky.

Nevertheless, even for gossip learning, there are robust parameter choices, for example, $A = 5, C = 10$. This parameter setting is also suitable for push gossip in all the settings we examined.

As a final note, let us compare the performance of our different algorithm variants. Even SIMPLE represents a significant improvement over the proactive approach, but GENERALIZED and RANDOMIZED outperform it robustly. Considering the best parameter settings, GENERALIZED has a slight advantage over RANDOMIZED in the push gossip application, and the reverse is true in gossip learning.

5.3.3 A Note on the Number of Tokens

Although we have a strong experimental focus, for completeness we present a short analytical derivation of the average number of tokens in the system. This property is interesting as the dynamics of the system depends on the available tokens. We assume a failure-free scenario. We use our previous notations, but here let $a(t)$ denote the average number of tokens over the nodes at time t and let $w(t)$ be the average number of messages sent (or, equivalently, received) by a node until time t. Now, we can write the mean-field model

$$\frac{da}{dt} = \frac{1}{\Delta} - \frac{dw}{dt} \tag{5.8}$$

$$\frac{d^2w}{dt^2} = \frac{dw}{dt}(\text{reactive}(a, u) - 1) + \frac{1}{\Delta}\text{proactive}(a) \tag{5.9}$$

The first equation states that a is increased by the constant rate of generated tokens (one per each cycle of length Δ) and decreased by the number of tokens used up. The second equation states that the change of the message sending rate is given by the number of reactive messages triggered by the incoming messages (also taking into account the fact that the one incoming message is "replaced" by the reactively generated messages triggered by it) and the number of proactive messages that are

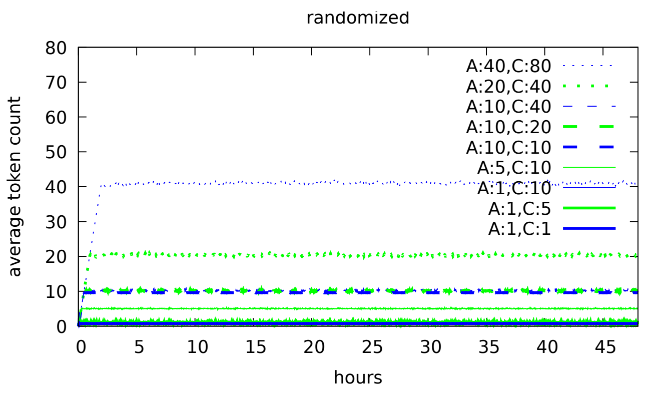

Figure 5.4: *Average number of tokens (gossip learning, failure free scenario).*

sent once in every cycle.

Now, assuming the equilibrium state when $da/dt = 0$ and $d^2w/dt^2 = 0$, solving the resulting equations gives us

$$1 = \text{reactive}(a, u) + \text{proactive}(a). \tag{5.10}$$

This can be solved for a for a fixed u. For the most promising version: randomized token account, solving the equation gives us $a = A \cdot C/(C + 1)$ for $u = 1$ (this means $a \approx A$). The assumption $u = 1$ is acceptable for gossip learning where most incoming messages are better than the locally stored random walk. Indeed, our validation runs (Figure 5.4) show a very good agreement with the predicted value.

5.4 Experimental Analysis of Partitioned Token Account

Here, we evalute partitioned token gossip learning in actual machine learning scenarios, using proactive gossip learning and federated learning (described in Section 4.2) as baselines.

Unlike the *continuous transfer* scenario examined in Chapter 4, here, we study the *bursty transfer* setup where all the nodes communicate only during a given percentage of the time. Without any modification, the behavior of federated and gossip learning is quite similar to their behavior in the continuous transfer scenario. Although gossip algorithms work slightly better due to the reduced number of parallel transfers, the bursty transfer scenario offers a possibility to implement specialized techniques that take advantage of bursty transfer explicitly.

In the case of gossip learning, we introduced the token account flow control technique, as described above.

In the case of federated learning, one such technique is when the master communicates only with a subset of the workers in each round, selecting a different subset each time. This way, although the workers communicate in a bursty fashion, the global model still evolves relatively fast.

5.4.1 Experimental Setup

The experimental setup is the same as in Section 4.3, with the differences noted below.

We use 2-day segments from the smartphone trace, but we use only the second 24-hour period for learning; the first day is used for achieving a token distribution that reflects an ongoing application. This warm-up period can represent a previous, unrelated learning task executed on the same platform, or the sending of empty messages; it does not count towards the communication costs. For fair comparison, we use the same period for learning also in the case of algorithms that do not use tokens.

The cycle length parameters Δ_g and Δ_f were set in a different way. We assume that we transfer data only during a given percentage of the time, say, 1% of the time. Let δ denote the transfer time and let $p \in (0, 1]$ be the proportion of the time we use for transfer. Here, we set the gossip cycle length $\Delta_g = \delta/p$. To implement the bursty model in federated learning, we have many choices for the cycle length depending on how many nodes the master wants to contact in a single cycle. If we set $\Delta_f = (\delta_{up} + \delta_{down})/p$ (where δ_{up} and δ_{down} are the upload and download transfer times, respectively), then the master should contact all the nodes as before so the only effect is the slowdown of the algorithm. If we set a shorter cycle length then the master will contact only a subset of the nodes to achieve the required proportion of p overall. We will examine this latter case when the cycle length is set so that 1% of the nodes are contacted in a cycle: $\Delta_f = \delta_{up} + \delta_{down}$, where we need $p = 1/100$ to hold as well. This way, the master communicates continuously while the nodes communicate in bursts. When compression is used, δ_{up} and δ_{down} might differ.

On average, the two algorithms still transfer the same number of bits overall in the network during the same amount of time. Furthermore, continuous transfer is the special case of bursty transfer with $p = 1$.

For gossip learning, we used the randomized token account implementation with parameters $A = 10$ and $C = 20$, based on our results in Section 5.3.2.

5.4.2 Partitioning

Model partitioning is especially helpful for token gossip learning, however, this technique has other advantages as well. To verify this, we compared partitioned and non-partitioned variants in several scenarios (Figure 5.5). We show the scenario where the effect in question is the clearest. Clearly, the partitioned implementations con-

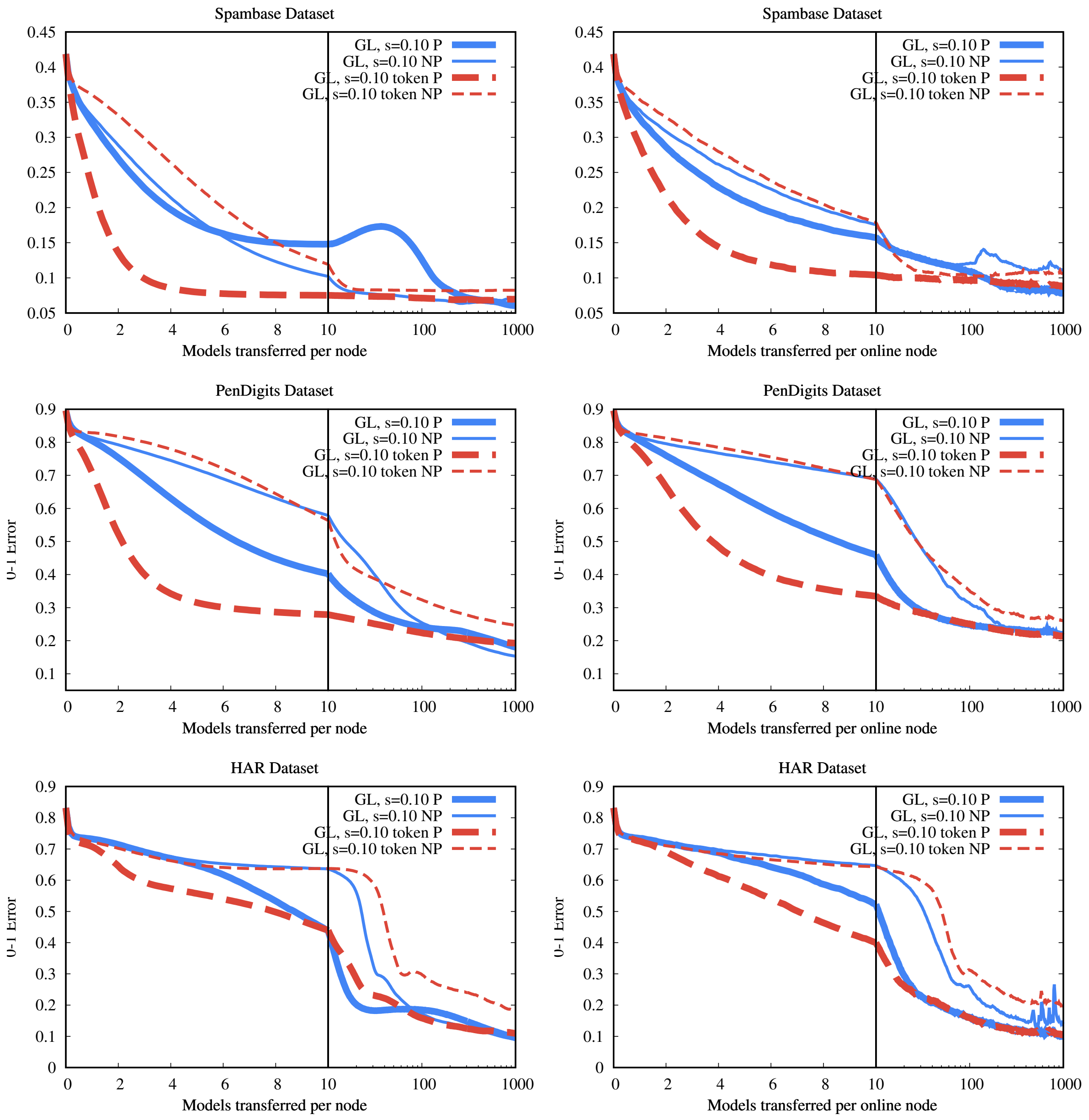

Figure 5.5: *Gossip learning with one node for each example, bursty transfer, subsampling probability $s = 0.1$, no-failure (left) and smartphone trace with long transfer time (right). Variants with and without model partitioning are indicated by P and NP, respectively.*

sistently outperform the non-partitioned ones, although in the no-failure scenario, classical gossip learning appears to suffer a temporary setback during convergence. Note that this is a case where the hyperparameters are not exactly optimal, as we explained in Section 4.3.3 (Figure 4.2).

The improved performance in the partitioned case is due to the more fine-grained handling of the age parameter. Recall that in the partitioned implementation, all the partitions have their own age and are updated accordingly. In the smartphone trace scenario, this feature is especially useful, since when a node comes back online after an offline period, its model is outdated. During the first merge operation on the model, only those parameters will get an updated age parameter that were actually updated, that is, those that are included in the merged partition. Without partitioning, only a random subset of the parameters will be merged, but the entire model will get a new age value. This is a problem because in the next merge operation that they are included in, the weight of these old parameters will be too large. Because of this, from now on, all the experiments are carried out with model partitioning, and this fact will not be explicitly indicated.

5.4.3 Small Scale

Figure 5.6 shows the performance in the bursty transfer scenario. Clearly, the convergence of each algorithm becomes faster than in the continuous communication case (Figure 4.5). This suggests that it is better to allow for short but high bandwidth bursts as opposed to long but low bandwidth continuous communication. We can also observe that token gossip converges faster than regular gossip in most cases. Also, the best gossip variant is, again, competitive with the federated learning algorithm.

5.4.4 Large Scale

In the large scale scenario, the number of examples per node was the same as in the 100-node scenario, but the network size equaled the size of the database. The results are shown in Figure 5.7. (For easier comparison, we also include the plots of the continuous transfer scenario from the previous chapter.)

The first observation we can make is that in the bursty transfer scenario faster convergence can be achieved. This is due to the algorithms that exploit burstiness.

We can see that the best gossip variants are still competitive with the best federated learning variants.

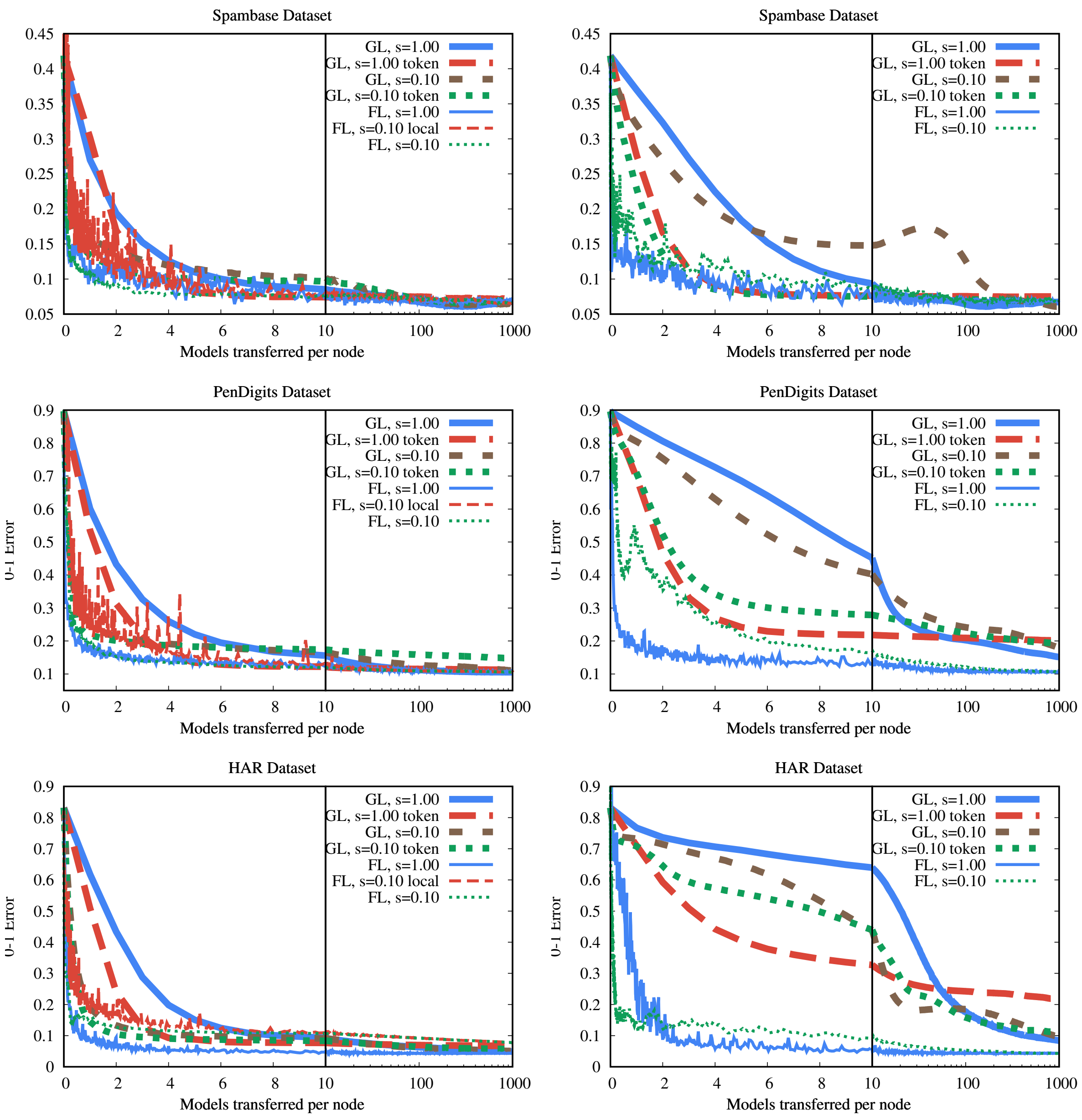

Figure 5.6: *Federated learning and gossip learning with 100 nodes (left) and with one node for each sample (right), no-failure scenario, in the bursty transfer scenario.*

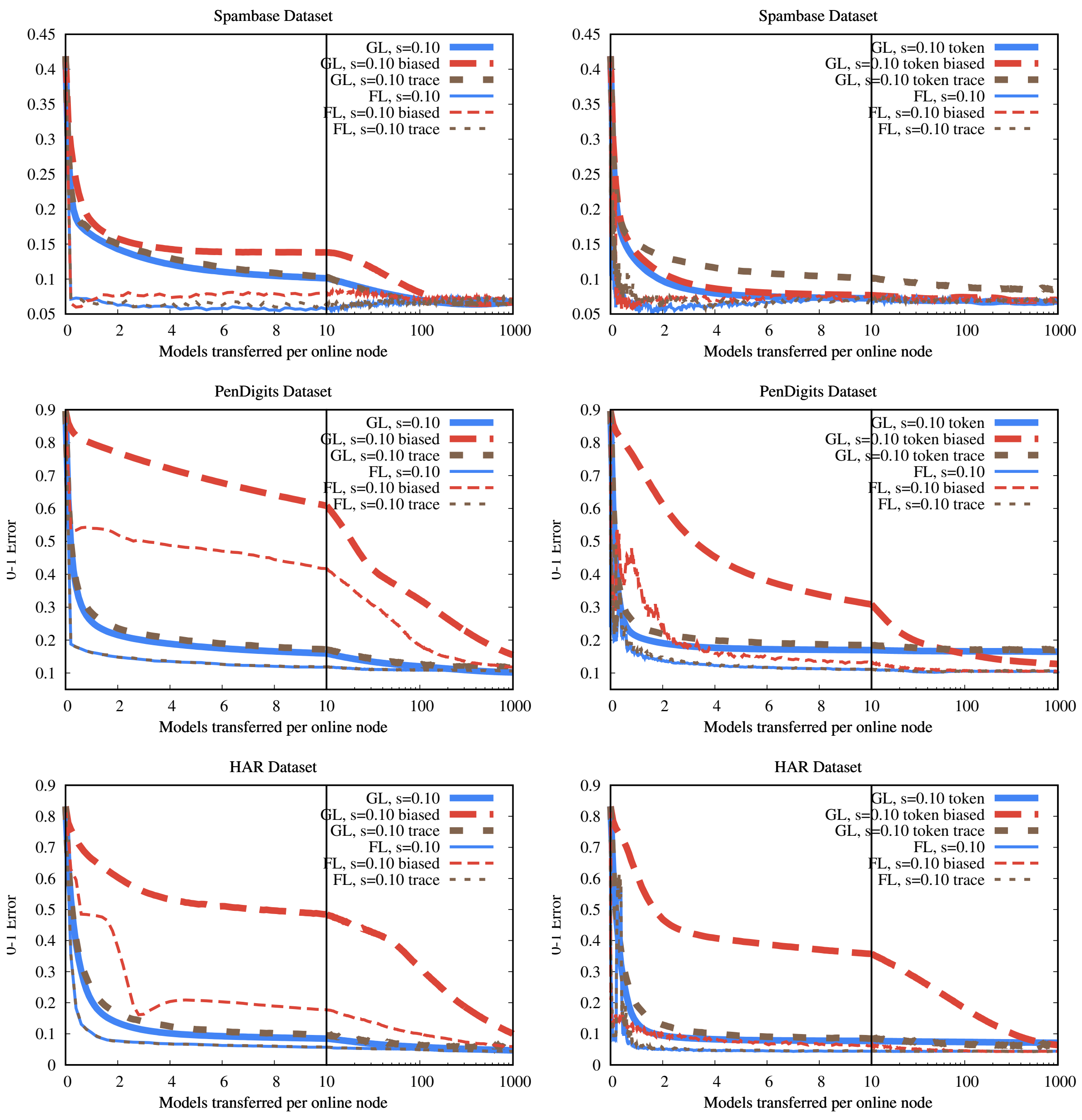

Figure 5.7: *Selected experiments in the large scale scenario. Continuous transfer (left) and bursty transfer (right). 'Biased' indicates single class assignment, 'trace' indicates the smartphone trace scenario.*

5.5 Related Work

Raghavan et al. [67] used a gossip protocol to implement a distributed token bucket limiter, where the goal was to control the global aggregate traffic through the cooperation of individual rate limiters. This is orthogonal to our work, because we wish to control the local traffic at all the individual nodes, while at the same time we wish to optimize a global application-specific performance measure, such as speed of convergence.

Rodrigues et al. [70] used token buckets as part of their adaptive broadcast solution. There, the emission rates were adaptive and the token bucket was used to control the input rate, that is, the rate at which a node accepts new events to broadcast. The gossip protocol itself was purely proactive, thus the efficiency of the broadcast under a fixed cost (the focus of our work) was not addressed. Frey et al. [33] applied token bucket rate limiting over the upload links to evaluate the effect of limited bandwidth, but other options for rate limiting were not investigated.

Wolff et al. [84] present a distributed data mining approach based on a decentralized algorithm to test whether the Euclidean norm of the average of vectors is within a threshold. They apply a leaky bucket algorithm for rate limiting, which makes their system periodic (thus, proactive). Here, a significant improvement in convergence speed could be expected using simple token bucket algorithms instead, and further optimization using our various token account algorithms may be possible.

Another application area is decentralized replication schemes where the dominant approach used to be reactive (for example, replicate when the number of replicas is below a threshold). First, Sit et al. [77] proposed a fully proactive scheme to deal with bursts, and this was followed by several hybrid proactive/reactive systems. For example, Duminuco et al. [29] proposed an adaptive version of the proactive scheme as well as a hybrid scheme that switches to purely reactive operation when the availability of data is critically low. Controlling the available repair-budget with the help of a token account method is a promising approach in this area as well.

5.6 Discussion and Conclusions

In this chapter, we introduced the token account service that serves as a communication layer for a large class of decentralized applications. This class includes asynchronous decentralized message passing applications such as gossip broadcast, gossip-based machine learning, and chaotic iteration methods. Any decentralized protocol might benefit from the service that is based on some form of periodic proactive local communication.

The main motivation was that we wanted to combine the advantages of proactive and reactive communication models. The reactive communication model has a crucial advantage: it often results in very fast convergence in several different applications. This is because nodes react immediately to new information, there is no idle time. However, since the number of messages is not controlled explicitly, they can generate too many or too few messages. Too many messages may be generated due to cascading instantaneous reactions to propagating new information. But too few messages can also be generated since messages are sent only in response to other messages. If some of the messages are never delivered due to failures or due to application specific filters, the overall amount of communication can decrease and the system might even arrive at a complete standstill.

The proactive communication model controls the number of messages explicitly, but it often results in an inferior convergence speed due to sitting on new information until the next round starts.

Token account algorithms were demonstrated here to maintain a very tight control over the number of messages we send, yet they were also shown to achieve a very significant speedup relative to a purely proactive implementation. In the case of gossip learning, we saw that the token account algorithm approximates the speed of a “hot potato” random walk, when the walk wastes no time at any of the nodes. In the case of the push gossip application, the delay of receiving the freshest update is one third of that of the proactive implementation. We achieved a significant speedup even in the case of chaotic iteration.

To evaluate this token-based flow control technique with the mergeable version of gossip learning, we performed machine learning over three different datasets. We also introduced a partitioned variant of the token account algorithm, to properly make use of sampling-based compression. Our results confirmed that the token-based flow control approach outperforms proactive gossip learning. Furthermore, it could achieve a performance comparable to federated learning in the random label assignment scenarios.

However, to achieve these results, the compression mechanism must be based on partitioning, as opposed to simple subsampling. The reason is that this way, the different partitions can form “hot potato” chains separately, whereas with subsampling, these chains cannot form because sampling picks different weights in every step.

In general, our results indicate that it is better to allow nodes to communicate in bursts (maximal bandwidth for a short time) than set a low bandwidth cap while allowing for continuous communication.

Contribution

In this chapter, the contributions of the author were: the design and evaluation of the token account algorithm; the analytical derivation of the average number of tokens in the system; and the design of the partitioned token gossip learning algorithm.

Chapter 6

Gossip Learning Using Compressed Averaging

Mean estimation has been studied in decentralized computing for a long time [13, 44, 48, 86]. The applications of these algorithms include data fusion in sensor networks [87], distributed control [63] and distributed data mining [81]. A very interesting potential new application is federated learning, where a deep neural network (DNN) model is trained on each node and these models are then averaged centrally [58]. This average computation could be decentralized, allowing for fully decentralized solutions, such as gossip learning [64], that are promising candidates to support applications for the common good [15]. The reason is that these solutions can be deployed without any investment at all, relying only on user devices and no additional infrastructure, without any pressure to make a profit. However, since DNNs may contain millions of floating-point parameters all of which have to be averaged simultaneously, optimizing the utilized bandwidth during decentralized averaging becomes the central problem.

Many approaches have been proposed for bandwidth-efficient average calculation. For example, floating point numbers can be compressed to a few bits using different quantization methods and these quantized values can then be averaged by a server [50, 79]. This is a synchronized and centralized solution, and the approach also introduces an estimation error. Quantization has been studied also in decentralized gossip protocols where the communicated values are quantized onto a fixed discrete range (see, for example, [90]). Here, an approximation error is introduced again, even in reliable networks, and message exchanges cannot overlap in time between any pairs of nodes.

In control theory, more sophisticated dynamic quantization approaches have been proposed that can provide exact convergence at least in reliable systems by compensating for the quantization error. An example is the work of Li et al. [53]. Here, full synchronization and reliability are assumed, and the quantization range is scaled by a fixed scaling function. Dynamic quantization has also been proposed in the context of linear control in general, again, in a synchronized model [34]. Carli et al. [17] adopt the compensating idea in [53] and compare it with other static (non-adaptive) quantization techniques. The same authors also study adaptive quantization; that is, dynamically changing the sensitivity of the quantizer [16] (originally proposed in [14]), which is feasible over a fixed communication overlay. The system model in these studies assumes reliability and atomic communication as well.

A rather different kind of method involves compressing a stream of floating point values using prediction and leading zero count compression [69]. Although this method could be adapted to our application scenario with some modifications, in this chapter we focus only on the quantization-based compression methods.

Our contributions include a modified push-pull averaging algorithm and a novel codec. These two contributions are orthogonal: the codec can be used along with any algorithm and the push-pull algorithm can use any codec. The novel codec, called *pivot codec*, encodes every floating point value onto a single bit and it can adapt dynamically to the range of the encoded values. The novel push-pull protocol is robust against message drop failure, it does not require the synchronization of the clocks of the nodes, and it includes a smoothing feature based on recorded link-flows that improves the performance of our compression codec. Furthermore, we propose a novel variant of gossip learning that uses this codec-based compression to achieve a higher communication efficiency than previous methods based on subsampling. Koloskova et al. have a similar focus but they apply only simple stateless quantization [49].

We evaluate our contributions in simulation. We compare our solutions with the competing codecs and algorithms from related work and show that we can improve both robustness and the compression rate significantly. Among our machine learning experiments we also include a transfer learning scenario, thereby demonstrating that it is feasible to adapt pre-existing deep neural network models to another domain by training only their last layer, which makes them accessible for gossip learning applications.

6.1 Background

6.1.1 System Model

Our system model is similar to that described in Section 2.2. However, our protocols assume that the neighbor set is stable, and the delay of most (but not necessarily all) of the messages that are delivered is less than an upper bound. This upper bound is at least half of the gossip period, or more, depending on the overlay network.

In the context of the problem of average consesus, we do not consider node failure, but assume that messages can be lost and their order of delivery is not guaranteed.

6.1.2 Codec Basics

Central to our algorithms is the concept of encoding and decoding messages over a given directed link using a codec. A codec consists of an encoder and a decoder placed at the origin and the target of the link, respectively. We assume that the link is used to send a series of real valued messages during the execution of the protocol. We follow the notations used in [60]. First of all, the compression (or encoding) is based on quantization, that is, mapping real values to a typically small discrete space (an alphabet) denoted by S. The decoding maps an element of alphabet S back to a real value.

Codecs may also have state. This state might contain, for example, information about the current granularity or scale of the encoding, the previous value transmitted and elapsed time. The state space will be denoted by Ξ. Every codec implementation defines its own state space Ξ (if the implementation is stateful). Both the encoder and the decoder are assumed to share the same state space.

We now introduce a notation for the mapping functions mentioned above. Let $Q : \Xi \times \mathbb{R} \to S$ denote the encoder (or quantizer) function that maps a given real value to a quantized encoding based on the current local state of the encoder. Let $K : \Xi \times S \to \mathbb{R}$ denote the decoding function that maps the encoded value back to a real value based on the current local state of the decoder. Finally, let $F : \Xi \times S \to \Xi$ define the state transition function that determines the dynamics of the state of the encoder and the decoder. Note that in a given codec both the encoder and the decoder uses the same F. These three mappings are always executed in tandem, that is, an encoded message is decoded and then the state transition is computed.

Although the encoder and the decoder are two remote agents that communicate over a limited link, the algorithms we discuss will ensure that both of them maintain an identical state. In this sense, we can talk about the state of the codec. To achieve this, first we have to initialize the state using the same value ξ_0. Second, if the encoder and the decoder have identical states at some point in time, then an identical state can be maintained also after the next transmission, because the encoder can simulate the decoder locally, thus they can both execute the state transition function with identical inputs. Note that here we assumed that communication is reliable. If this is not the case, the algorithms using the codec must handle unreliability appropriately so as to maintain the identical states.

6.2 Proposed Algorithms

We first discuss our novel codec and then present the modified push-pull averaging protocol in several steps, addressing its robustness, compression, and smoothing features. Finally, we adapt this protocol for machine learning.

6.2.1 Pivot Codec

Here we describe our codec implementation that we coined the *pivot codec*, for reasons that will be explained below. The main goal in our implementation was aggressive compression, so we put only a single bit on the wire for each encoded value. This means $S_{pivot} = \{0, 1\}$.

The intuition behind the design is that we treat the encoder and the decoder as two agents, such that the encoder stores a constant value and the decoder has to guess this value based on a series of encoded messages. Obviously, in real applications the encoded value is rarely constant. However, the design is still efficient if the encoded values do not change much between two transmissions. In fact, this assumption holds in many applications, including decentralized mean approximation, which allows for an efficient compression. Many competing codecs, especially simple quantization techniques, do not make any assumptions about the correlation of subsequent encoded values, hence they are unable to take advantage of the strong positive correlation that is present in many applications.

The codec is stateful. The state is defined by a triple $(\widehat{x}, d, s_{last}) \in \Xi_{pivot} = \mathbb{R} \times \mathbb{R} \times S_{pivot}$. Here, $\widehat{x}$ is the approximation of the *pivotal value*, namely the actual (constant or slowly changing) real value stored by the encoder agent. The remaining values are d, the signed step size, and s_{last}, the last encoded value that was transmitted. The encoding function is given by

$$Q_{pivot}((\widehat{x}, d, s_{last}), x) = \begin{cases} 1, & \text{if } |\widehat{x} + d - x| < |\widehat{x} - x| \\ 0, & \text{otherwise,} \end{cases} \tag{6.1}$$

where x is the value to be encoded. In other words, the encoded value is 1 if and only if adding the current step size to the approximation makes the approximation better. Accordingly, the decoding function

$$K_{pivot}((\widehat{x}, d, s_{last}), s) = \begin{cases} \widehat{x} + d, & \text{if } s = 1 \\ \widehat{x}, & \text{otherwise} \end{cases} \tag{6.2}$$

will add the step size to the current approximation if and only if a 1 is received. Note that this design ensures that the approximation never gets worse. It can only get better or stay unchanged, assuming the encoded value is a constant. Note that both the encoder and the decoder share the same state. This is possible because the encoder can simulate the decoder locally, thus both the encoder and the decoder can compute the same state transition function given by

$$F_{pivot}((\widehat{x}, d, s_{last}), s) = \begin{cases} (\widehat{x} + d, 2d, s), & \text{if } s = 1 \wedge s_{last} = 1 \\ (\widehat{x} + d, d, s), & \text{if } s = 1 \wedge s_{last} = 0 \\ (\widehat{x}, -d/2, s), & \text{otherwise.} \end{cases} \tag{6.3}$$

Here, if d is added for the second time, we double it (assuming that the direction is good) and if we have $s = 0$ then we halve the step size and reverse its direction, assuming that adding d overshot the target. The step size is left unchanged after its first successful application (middle line).

In order for the encoder and the decoder to share their state, they also have to be initialized identically. The initial state ξ_0 might use prior knowledge, for example, prior information about the expected mean and the variance of the data are good starting points for $\widehat{x}$ and d, respectively, but a generic value like $\xi_0 = (0, 1, 0)$ can also be used.

6.2.2 Robust Push-Pull Averaging

As a first step towards the compressed algorithm, here we propose a variant of push-pull averaging (Algorithm 6.1) that is robust to message loss and delay and that also allows for the application of codecs later on. The algorithm is local, hence the scope of the variables is limited to the current node. We assume that the links are directed. This means that if both $A \to B$ and $B \to A$ exist, they are independent links. Over a given directed link there is a series of attempted push-pull exchanges with push messages flowing along the link and the answers (pull messages) moving in the opposite direction. The algorithm ensures that both ends of each link will eventually agree on the flow over the link. This will ensure a sum preservation (also called mass conservation) property which we prove below.

The algorithm is similar to traditional push-pull averaging in that the nodes exchange their values first. However, as a generalization the new value will not be the average of the two values, but instead a difference δ is computed at both sides using a "greediness" parameter $H \in (0, 1]$, where δ can be viewed as the amount of material being transferred by the given push-pull exchange. Note that both sides can compute the same δ (with opposite signs) independently as they both know the two raw values and they have the same parameter H. Here, $H = 1$ results in the traditional variant, and smaller values allow for stabilizing convergence when the push-pull exchanges are not atomic, in which case—despite sum-preservation—convergence is not guaranteed.

As for ensuring sum preservation, we assign an increasing unique ID to all push-pull exchanges. Using these IDs we simply drop out-of-order push messages. Dropping push messages has no effect on the update counters and the local approximations so no further repair action is needed. When the pull message arrives in time, the update is performed, and since the sender of the pull message (say, node B) has already performed the same identical update (using the same δ), the state of the network is consistent. If, however, the pull message was dropped or delayed then the update performed by node B has to be rolled back. This is done when B receives the next push message and learns (with the help of the update counters) that its previous pull message had not been received in time. The update can be rolled back using δ, which ensures that the sum in the network is preserved.

After this intuitive explanation, let us describe the sum-preservation property in formal terms. For this, let us assume that there exists a time t after which there are no failures (message drop or delay). We will show that after time t the sum of the approximations will eventually be the same as the original sum of local values.

Definition 2. *We say that, over link $A \to B$, a* successful transaction *with ID j is completed when node A receives a pull message with $id = j$ from node B before sending the next push message with $id = j + 1$ to B.*

Let j_k be the ID of the kth successful transaction over link $A \to B$, and let $j_0 = 0$. For any variable v of Algorithm 6.1, let v^X denote the value of variable v at node X.

Theorem 2. *For any index $K \geq 0$, right after processing the pull message from B to A of a successful transaction j_K (or for $K = 0$ right after initialization), A and B agree on the total amount of mass transferred over the link $A \to B$, furthermore, $u^A_{B,out} = u^B_{A,in} = K$ holds.*

Proof. The theorem trivially holds for $K = 0$. Assume that the theorem holds for $K = k - 1$. We show that it holds for $K = k$ as well. First of all, line 29 is executed if and only if the transaction is successful. Then, $u^A_{B,out}$ is incremented by 1, therefore $u^A_{B,out} = k$ indeed holds right after the kth successful transaction. As for $u^B_{A,in}$, the inductive assumption states that $u^B_{A,in} = k - 1$ right after the $(k-1)$-th successful transaction. After this point, there will be a series of incoming push messages that are not out of order with IDs $i_1, \ldots, i_n$ such that $j_{k-1} < i_1 < \cdots < i_n = j_k$, where j_k is the ID of the kth successful transaction. These incoming messages are assumed to be processed sequentially. In all of these push messages we will have $u = k - 1$. It follows that after processing i_1 we will have $u^B_{A,in} = k$ and after processing each new message $i_2, \ldots, i_n$ we will still have $u^B_{A,in} = k$. This means we have $u^A_{B,out} = u^B_{A,in} = k$ right after the successful transaction j_k.

Let us turn to the transferred mass, and show that after the kth successful transaction A and B will add or remove, respectively, the same δ mass from their current approximations. This is analogous to our previous reasoning about the counters $u^A_{B,out}$ and $u^B_{A,in}$, exploiting the observation that only at most one update has to be rolled back between consecutive updates (which can be done due to recording $\delta^B_{A,in}$) until the correct update occurs. Also, due to recording s^A_B both A and B can compute the same δ despite the delay at A between sending the push message and updating after receiving the pull message. □

Corollary 1. *After time t push-pull exchanges become atomic transactions so after a new push message is sent on each link, each pair of nodes will agree on the transferred amount of mass, resulting in global mass conservation. Also, the algorithm will become equivalent to the atomic push-pull averaging (for $H = 1$), for which convergence has also been shown [44].*

Note that if the message delay is much longer than the gossip period Δ then progress becomes almost impossible, because sending a new push message over a link will often happen sooner than the arrival of the pull message (the reply to the previous push message), so the pull message will be dropped. Therefore, the gossip period should be longer than the average delay. In particular, if the gossip period is at least twice as large as the maximal message delay then no pull messages will be dropped due to delay.

Transactions over different links are allowed to overlap in time. When this happens, it is possible that the variance of the values will temporarily increase, although the sum of the values will remain constant. In networks where transactions overlap to a great degree, it is advisable to set the parameter H to a lower value to increase stability.

6.2.3 Compressed Push-Pull Averaging

Here, we describe the compressed variant of push-pull averaging, as shown in Algorithm 6.2. Although the algorithm is very similar to Algorithm 6.1, we still present the full pseudocode for clarity. Let us first ignore all the f variables. The algorithm is still correct without keeping track of the f values, these are needed to achieve a smoothing effect that we explain later on. Without the f values, the algorithm is best understood as a compressed variant of Algorithm 6.1 where values are encoded before sending and decoded after reception. There are some small but important additional details that we explain shortly.

In the messages, the value of x is compressed, but the u and id values are not. This is not an issue, however, because our main motivation is the application scenario where x is a large vector of real numbers. The amortized cost of transmitting two uncompressed integers can safely be ignored.

The algorithm works with any codec that is given by the definition of the state space Ξ, the alphabet S, and the functions Q, F and K, as described previously. We maintain a codec for every link and for every direction. That is, for every directed link (j, i) there is a codec for the direction $j \to i$ as well as $j \leftarrow i$. For the $j \to i$ direction, node j stores the codec state (used for encoding push messages) in $\xi_{i,out,loc}$ and for the $j \leftarrow i$ direction the codec (used for decoding pull messages) is stored in $\xi_{i,out,rem}$ at node j. In this notation, "out" means that the given codecs are associated with the outgoing link. The states for the incoming links are stored in a similar fashion.

Recall that codecs must have identical states at both ends of the link and this state is used for encoding and decoding as well. For example, the codec state $\xi_{i,out,loc}$ at node j for the direction $j \to i$ should be the same as $\xi_{j,in,rem}$ at node i. This requirement is implemented similarly to the calculation of δ in Algorithm 6.1. The codec state transitions, too, are calculated at both ends of each link independently, but based on shared information, so both nodes can follow the same state transition path, assuming also that the states have the same initial value ξ_0. This state transition is computed right after computing δ, in line 36.

Apart from δ, here we also need the previous codec states for rolling the last update back if a pull message was dropped or delayed. To this end, the codec states are backed up (line 21) and are rolled back when needed (line 18).

When calculating δ, we must take into account the fact that encoding and decoding typically introduces an error. Therefore, in order to make sure that both nodes compute the same δ, both nodes have to simulate the decoder at the other node, and work with the decoded value instead of the exact value that was sent (line 35). Fortunately, this can be done, since the state of the decoder at the other node can be tracked locally, as explained previously. However, since we are no longer working with the exact values, there is no guarantee that every update will actually reduce variance over the network, so it is advisable to set H to a value less than one.

6.2.4 Flow Compensation

So far we have ignored the f variables in Algorithm 6.2. The purpose of these variables is to make compression more efficient by making the transmitted values over the same link more similar to each other. This way, good stateful adaptive codecs can adjust their parameters to the right range achieving better compression.

The f values capture the flow over the given link. This approach was inspired by flow-based approaches to averaging to achieve robustness to message loss [47, 62]. However, our goal here is not to achieve robustness, but rather to reduce fluctuations in the transmitted values. The algorithm accumulates these flows for each link in both directions. In addition, the flow value is added to the transmitted value. This has a smoothing effect, because if a large δ value was computed over some link (that is, the value of x changed by a large amount), then the sum of x and the flow will still stay very similar the next time the link is used. The beneficial effect of this on compression will be demonstrated in our experimental study.

Clearly, both nodes can still compute the same δ locally, because the flow value is also known at both ends of a link, only the sign will differ. Hence we can apply the formula in line 35.

6.2.5 Compressed Push-Pull Learning

For the basics of machine learning, see Section 2.1. We will refer to the machine learning model update step as training, to avoid confusion with the averaging update.

Our compressed push-pull learning algorithm is based on the compressed push-pull averaging protocol. The nodes perodically train their model (given by the parameter vector w) on the local data, as well as perform distributed averaging of the models (weighted by the model age t, which is the number of examples the model was trained on).

When used without model training, the algorithm falls back to computing the average of the initial w vectors weighted by their respective initial t values. This is achieved by simultaneously computing the average of tw and that of t, since the quotient of these is the weighted average of w.

The pseudocode is shown in algorithms 6.3 and 6.4. Models are encoded before sending and decoded after being received. During a push-pull transaction, the nodes exchange their encoded models, then, based on the decoded models, a difference vector δ is computed on both sides that represents for each parameter the amount of mass being transferred in the push-pull exchange. Both nodes compute the same δ (with opposite signs), because they use only the information that was exchanged, ignoring the current, uncompressed local model w. The difference is scaled by $H/2$, where $H \in (0, 1]$ is the same greediness parameter discussed earlier. After decoding the models, the codec states are updated.

The techniques used for compression and ensuring sum preservation are largly unchanged. One difference worth mentioning, though, is that we omitted the flow compensation component, because it had a negative influence on the machine learning performance. We speculate that the training and averaging steps tend to have opposing effects on the model parameters at a given node, therefore an adaptive codec can make a better guess at the current value if it ignores the already transferred mass.

Recall that we are averaging tw (and t as well) across the network. During compression, however, we encode w instead of tw. This is because tw will surely not converge, but w might, which is beneficial for adaptive codecs. Since t is transmitted, the remote node can still compute an estimate for tw. In the messages, only the model w is compressed. When w is a large vector, the amortized cost of transmitting the other variables is negligible.

The algorithm works with any codec that is given by the definition of the state space Ξ, the alphabet S, and the functions Q, F and K, as described previously. We apply these functions on the model parameter vector: the operation is performed elementwise, each parameter having its own codec state.

6.3 Experiments for Average Consensus

We evaluate our average consensus algorithms in simulation using PeerSim [59]. Apart from the modified push-pull protocol presented here, we experiment with the synchronized version of average consensus, the most well-known algorithm in related work in connection with quantized communication. In addition, we study a set of codecs and combine these with the two algorithms (synchronized iteration and our push-pull gossip). This way, both the codecs and the algorithms can be compared, as well as their different combinations.

Synchronized average consensus is described, for example, in [13]. The idea in a nutshell is that—assuming the values of the nodes are stored in a vector $x(t)$ at time t—if the adjacency matrix A of the nodes is invertible and doubly stochastic then the iteration $x(t+1) = Ax(t)$ will converge to a vector in which all the elements are equal to the average of the original values. The distributed implementation of such an iteration requires strong synchronization. Quantized and compressed solutions in related work focus on such approaches, as well as slightly more relaxed versions where the adjacency matrix can be different in each iteration, but the different iterations can never overlap.

The codecs we test include simple *floating point quantization (F16, F32)* assuming a floating point representation of 16 and 32 bits (half and single precision, respectively). Here, the codec is stateless, and decoding is the identity mapping. Encoding involves finding the numerically closest floating point value.

We also include the *zoom in - zoom out codec (Zoom)* of Carli et al. [16]. We cannot present this codec in full detail due to lack of space, but the basic idea is that an m-level quantization is applied such that there is a quantizer mapping to $m-2$ equidistant points within the $[-1,1]$ interval and the values -1 and 1 are also possible levels used for mapping values that are outside the interval. The codec state also includes a dynamically changing scaling factor that scales this interval according to the values being transferred. This codec resembles the pivot codec we proposed, and to the best of our knowledge this is the state of the art dynamic adaptive codec. Note that the minimal number of quantization levels (or alphabet size) is 3, when $m = 3$. The codec has two additional parameters: $z_{in} \in (0,1)$ and $z_{out} > 1$. The first determines the zoom-in factor and the second is the zoom-out factor. We fix the setting $z_{out} = 2$ based on the recommendation of the authors and our own preliminary results.

6.3.1 Experimental Setup

The network size is $N = 5{,}000$, and the results are the average of 5 runs. We also simulated a select subset of algorithms with $N = 500{,}000$ (single run) in order to demonstrate scalability. The overlay network is defined by a k-out network, where $k = 5$ or $k = 20$. In the case of synchronized average consensus, we transform this network into a doubly stochastic adjacency matrix A by dropping directionality and setting the weights on the links using the well-known Metropolis-Hastings algorithm: $A_{ij} = 1/(1+\max(d_i, d_j))$, where d_i is the degree of node i. Loop edges are also added with weight $A_{ii} = 1 - \sum_{j \neq i} A_{ij}$.

The initial distribution of values is given by the worst case scenario when one node has a value of 1, and all the other nodes have 0. This way, the true average is $1/N$ (where N is the network size). Our performance metric is the mean squared distance from the true average. We study the mean squared error as a function of the number of bits that are transferred by an average node to average a single value. Recall that we assume that many values are averaged simultaneously (we work with a large vector) so network latency can be ignored. This means that the number of transmitted bits can be converted into wall-clock time if one fixes a common bandwidth value for all the nodes.

We examine the value of the parameter H (see Algorithm 6.1) using a range depending on the actual codec (we determined the optimal value for each scenario and experimented with neighboring values). We also vary the cycle length Δ. We experiment with short and long cycles. When using short cycles, the round-trip time of a message is assumed to be 98% of the cycle length. With long cycles, the round trip time is assumed to be only 2% of the cycle length. The motivation of looking at these two extreme scenarios is that in the latter case messages overlap to a much lesser extent than in the former case. Thus, we wish to demonstrate that our solutions are robust to short cycles. As for failures, we simulate message drop failure, where the message drop rate is either 0% or 5%.

6.3.2 Results

Figure 6.1 gives a comparison of the performance of different codecs when using our push-pull algorithm. The parameters were optimized for every codec using a grid search in the space $H \in \{2^0, 2^{-1}, \ldots, 2^{-4}\}$, $k \in \{5, 20\}$, $z_{in} \in \{0.35, 0.4, \ldots, 0.85\}$ and $m \in \{4, 8, 16\}$. In all the four scenarios shown on the plots, the best parameter settings were $H = 1/2$ and $k = 5$ for the pivot codec and $H = 1/4$, $k = 5$, $m = 4$, and $z_{in} = 0.55$ for the zooming codec. For the floating point codecs, $H = 1/2$ and $H = 1$ were the best for short and long cycles, respectively, and $k = 20$ was the best without message drop. With message drop, the floating point codecs are more stable with $k = 5$ but they converge slightly faster with $k = 20$, especially with short cycles.

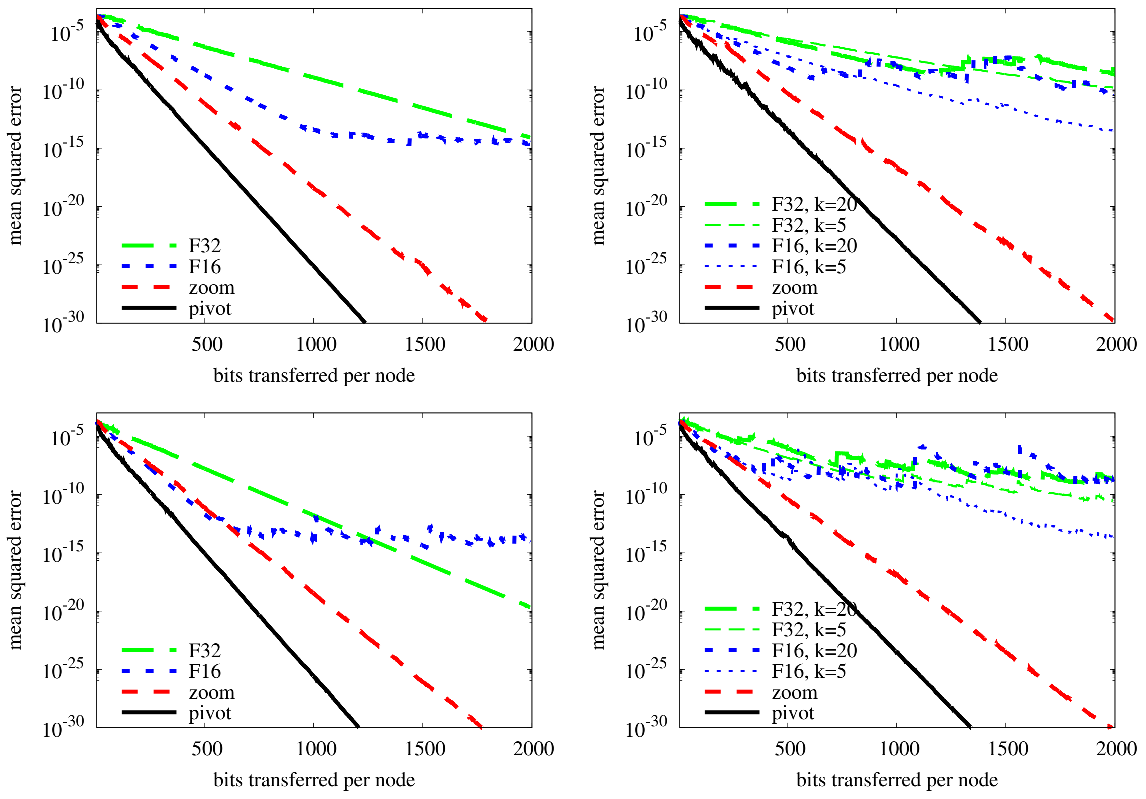

Figure 6.1: *Comparison of codecs in push-pull with no message drop (left) and a 5% message drop (right) with short cycles (top) and long cycles (bottom). The parameters of all of the codecs have been optimized.*

The pivot codec clearly dominates the other alternatives.

The difference between $k = 5$ and $k = 20$ is that in the former case more transactions are performed over a given fixed link. In the case of the stateless codecs, this means that $k = 5$ results in a more stable convergence because errors are corrected faster, but with $k = 20$ the correlation between consecutive updates over a fixed link are lower which results in a faster initial convergence. In the case of the pivot codec, Figure 6.2 illustrates the effect of parameters H and k. It is clear that the algorithm is robust to H, however, parameter k has a significant effect. Unlike the stateless codecs, the pivot codec benefits from a somewhat larger correlation between updates as well as the higher frequency of the updates over a link since these allow for a better prediction of the value at the other end of the link. The zooming codec has a similar behavior (not shown), and we predict that every adaptive codec prefers smaller neighborhoods.

Figure 6.3 presents a similar comparison using the synchronized average consensus algorithm. Note that here, the long and short cycle variants behave identically. Again, the parameters were optimized for every codec and the best parameter set-

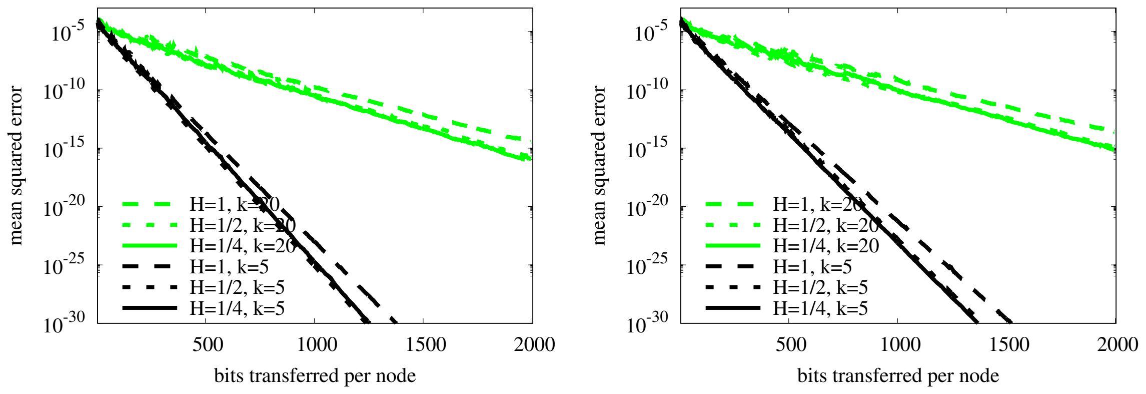

Figure 6.2: *The effect of parameters H and neighborhood size k on the pivot codec, with no message drop (left) and a 5% message drop (right).*

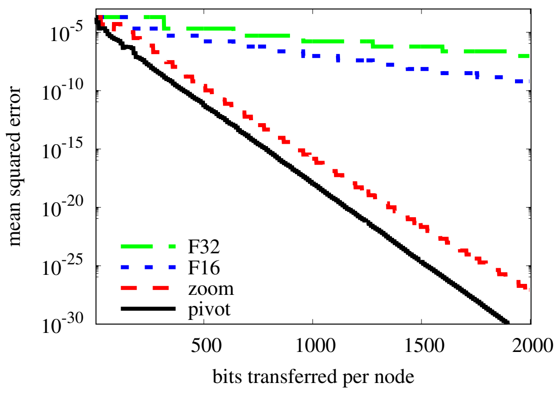

Figure 6.3: *Comparison of codecs in synchronized average consensus. The parameters of all of the codecs are optimized.*

tings were $H = 1/2$ and $k = 5$ for the pivot codec, $H = 1$ and $k = 5$ for the floating point codecs, and $H = 1$, $k = 5$, $m = 8$, and $z_{in} = 0.45$ for the zooming codec. Again, the pivot codec dominates the other alternatives. Furthermore, note that, for the pivot codec, the optimal parameters are the same as those in the case of the push-pull algorithm. This suggests that these parameters are robust.

Figures 6.1 and 6.3 allow us to compare the push-pull algorithm with the synchronized algorithm. It is clear that all the codecs perform better with push-pull than with the synchronized algorithm. This implies that the push-pull algorithm is a better choice for compression, independently of the selected codec.

Figure 6.4 contains two remaining observations. First, it demonstrates that the mean squared error of push-pull gossip does not depend on network size as the results with $N = 500{,}000$ (left plot) are very similar to those with $N = 5{,}000$ (Figure 6.1, top left). This is not surprising as this is predicted by theory when no compression is

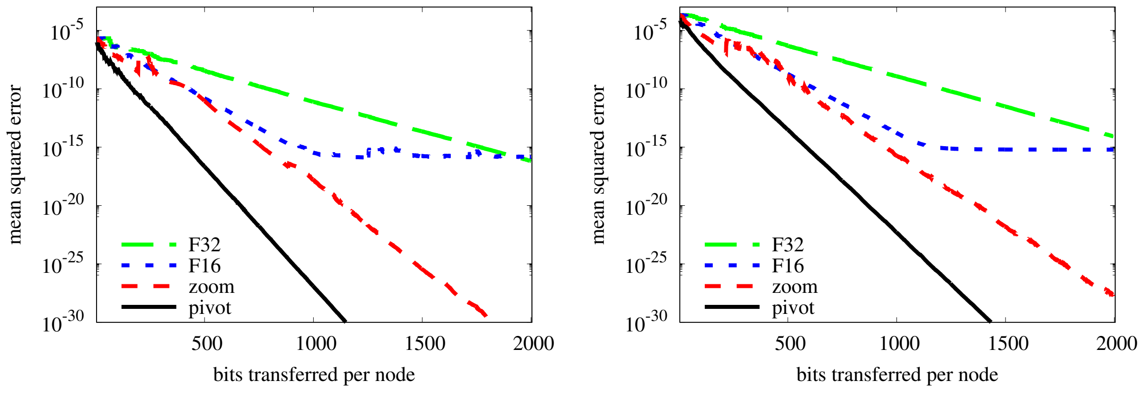

Figure 6.4: *Comparison of codecs with network size* $N = 500{,}000$ *(left) and without the flow compensation technique (with* $N = 5{,}000$*, right).*

applied [44].

Second, Figure 6.4 (right) shows the effect of the flow compensation technique introduced in Algorithm 6.2, where we used the f variables to smooth the stream of values over each link. As before, we optimized the parameters for all the codecs. The optimal parameter value for the pivot codec turned out to be $H = 1/8$ and $k = 5$. This means that if we drastically reduce H, thus smoothing the transactions much more aggressively with this alternative technique, the pivot codec still dominates the other codecs. However, we are not able to get the same compression rate we could achieve with flow compensation (Figure 6.1) so the flow compensation technique is a valuable addition to the protocol. The other codecs have the same optimal parameters as with flow compensation. Note that the zooming codec also benefits from flow compensation, although to a lesser extent. We also observed that the zooming codec is very sensitive to z_{in} in this case, small deviations from the optimal value result in a dramatic performance loss (not shown).

6.4 Machine Learning Experiments

Now, we shall describe our experimental setup and our results for compressed push-pull learning. We used partitioned token gossip learning as our baseline (see Section 5.2.3).

In our experiments we modeled the churn of the nodes using the trace described in Section 2.4. For a successful message transfer, both sides must stay online for the duration of the transfer. When choosing a random neighbor, only online nodes were considered.

Table 6.1: *Data set properties*

	HAR	MNIST	FMNIST
Training size	7352	60000	60000
Test size	2947	10000	10000
#features	561	784	784
#classes	6	10	10
Label distribution	$\approx$ uniform	$\approx$ uniform	$\approx$ uniform

6.4.1 Datasets

We used two different datesets to evaluate our algorithm, and a third dataset used for our transfer learning approach, as we describe in the next section. The main properties are shown in Table 6.1. The HAR (Human Activity Recognition Using Smartphones) database [2, 4] contains records that represent movements from 6 different classes (walking, walking upstairs, walking downstairs, sitting, standing, laying). The data was collected from the smart phones of 30 different people, using the accelerometer, gyroscope and angular velocity sensors. High level features were extracted based on the frequency domain.

The other dataset we used for evaluation is MNIST [52]. It contains images of handwritten digits with dimension 28×28, each pixel from the range $[0, 255]$. The Fashion-MNIST [85] dataset was used for transfer learning. It has the same parameters but it contains images of clothes and accessories instead of numbers.

6.4.2 Transfer Learning

In the case of our image recognition task, MNIST, we did not learn over the raw data directly but instead performed transfer learning [76], as we explain here. The idea is that we build a complex convolutional neural network (CNN) model offline over Fashion-MNIST and, before learning begins in the P2P network, all the nodes receive this pre-trained network. The nodes then use features extracted by this network to build a simple linear model over a different problem, namely MNIST. This way, we can learn (or, rather, fine-tune) a complex model with relatively little communication.

The CNN model for Fashion-MNIST had a LeNet-5-like architecture [52]. The layers were the following:

- 2D convolution $(6 \times 5 \times 5)$,
- 2D max-pooling (2×2),
- 2D convolution $(16 \times 5 \times 5)$,
- 2D max-pooling (2×2),

- dense layer with 120 units,
- dense layer with 84 units,
- classification layer with 10 units.

All the units in the layers use the relu activation function. After the training process, we removed the dense layers from the network. The last layer of this reduced model was used as the feature set for the MNIST dataset [76]. Fashion-MNIST has a more complex structure and represents images rich in detail, so the convolutional layers have to extract features that are potentially useful for other tasks as well. The extracted feature space has 400 dimensions as opposed to the original 784 features.

When training a linear model using these new 400 features over MNIST, the accuracy (the ratio of correct classifications over the test set) is 0.9785. When training the full CNN model over the raw MNIST dataset, the model can achieve an accuracy of 0.9890. At the same time, a linear model on the raw MNIST dataset just gives an accuracy of 0.9261. This clearly shows that transfer learning offers a significant advantage.

We reduced the number of features from 400 using Gaussian Random Projection [8]. With 128 features, the linear model has an accuracy of 0.9579. 78 features (about the 10% of the feature size of the original space) give us an accuracy of 0.9330. In our evaluation, we used the smallest feature space of 78 features.

6.4.3 Metaparameters

We used a fixed random k-out graph as the overlay network, with $k = 5$ or $k = 20$. The network size was 100. The training dataset was standardized (shifted and scaled so as to have a mean of 0 and variance of 1), and each example was assigned to one of these nodes.

For learning, we used logistic regression embedded in a one-vs-all meta-classifier, with a constant learning rate $\eta = 10^{-2}$ unless stated otherwise. We initialized both algorithms so that $(w, t) = \text{train}(\mathbf{0}, 0, D)$; that is, there is an initial training step.

We used the randomized token strategy (see Section 5.2.4) for the partitioned token gossip learning, with parameters $A = 10$, $B = 20$. The models were divided into 10 partitions, that is, a message contained (on average) 10% of the parameters. To make the baseline stronger, we assume, for the purposes of message size, that it encodes real numbers to a 16-bit floating point format. However, in the case of the baseline we do not actually perform the encoding; hence its performance will be an upper bound on any possible 16-bit floating point format, such as IEEE Half-precision Floating Point Format or Brain Floating Point Format. This means the baseline encodes a parameter to 1.6 bits per message on average.

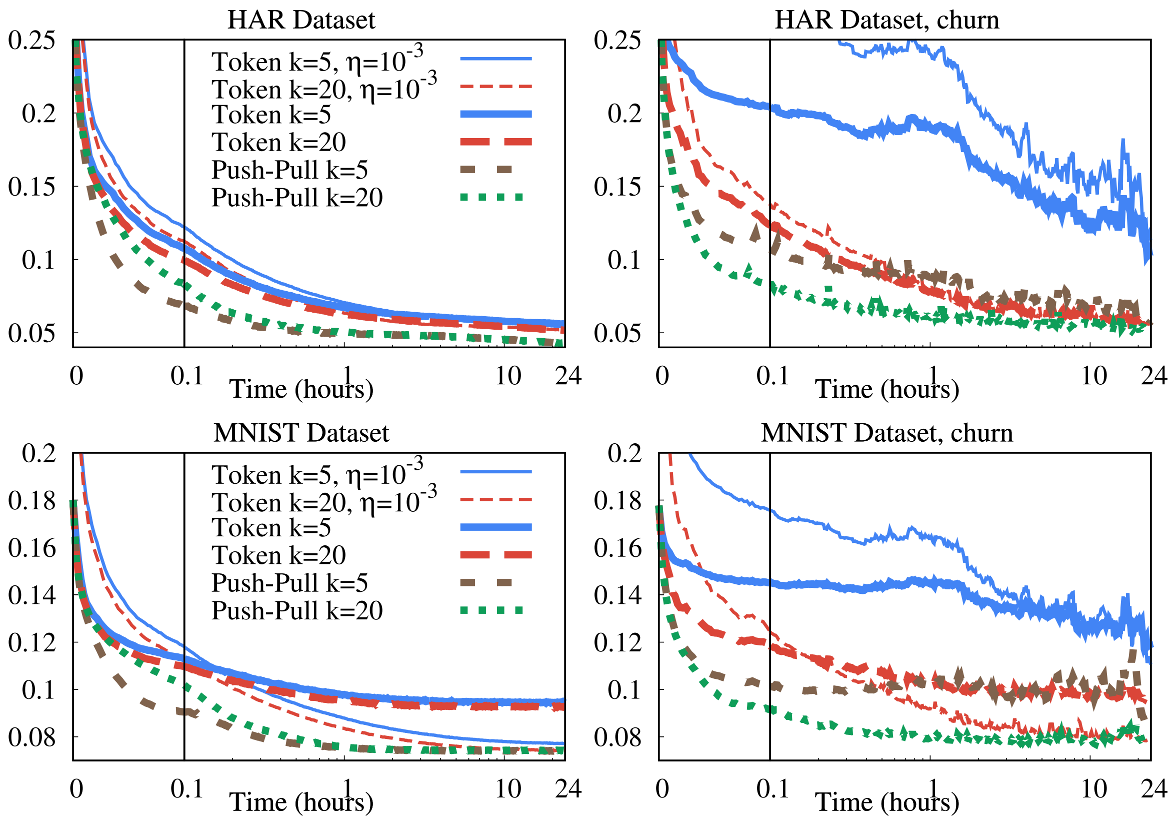

Figure 6.5: *Results over the HAR and MNIST datasets without and with churn.*

In the compressed push-pull learning experiments we used the greediness parameter $H = 0.5$ and the pivot codec, that encodes to a single bit. (Note that this means 2 bits of communication per parameter per cycle, since there are two messages per cycle on average.) We initialized its stepsize d to 10η. Our preliminary experiments suggested that this is a good setting in the case of constant learning rate and standardized datasets.

The length of the experiments was two simulated days. However, in the first 24 hours no training occurs, only dummy messages are sent; this period is used to “warm up” the token account algorithm to attain dynamics that reflect a continued use of the protocol. For example, when it is used as part of a decentralized machine learning platform that runs different learning tasks continuously. Only the second 24 hours are shown in the plots.

The cycle length of the baseline was set so that it could perform $10{,}000$ cycles in 24 hours. We set the cycle length of the push-pull algorithm so that on average, the two algorithms transfer the same number of bits during the same amount of time; this resulted in $8{,}000$ cycles.

The message transfer time of the baseline was set to one-hundredth of its cycle length, since such bursty communication benefits the token account algorithms. We set the transfer time for the push-pull algorithm to reflect the same bandwidth.

6.4.4 Results

The average 0-1 error over the online nodes on the test set as a function of time (or, equivalently, communication cost) is shown in Fig. 6.5. Note that the first part of the horizontal axis is linear, and the second part is logarithmic. Each plot is the average of 5 runs with different random seeds. The plots are noisy in the churn scenario, due to offline nodes with relatively poor models going online.

In the examined scenarios, compressed push-pull learning clearly outperformed token gossip learning, despite the latter’s benefits of lossless 16-bit compression and multiple learning rates. This can be seen by comparing how quickly the algorithms reach a certain level of error. In the no-churn scenario, on the HAR dataset, a 10% error is achieved by our novel algorithm in less than half, and a 6% error in less than one-ninth of the time needed by token gossip learning. On the MNIST dataset, a 10% error is achieved in less than one-fourth, and a 8% error in less than one-fifth of the time needed by token gossip learning.

Now, let us examine the effects of the out-degree k. Usually, a smaller k is worse, because it increases the mixing time of the graph. However, a bigger k results in less frequent communication over a given link; in the case of compressed push-pull with an adaptive codec, this makes the codec adapt slower, which can outweigh the mixing time. (In other words, more codecs require more communication to adapt.)

Still, an even more significant factor arises in the churn scenario: with $k = 5$, it is not uncommon that a node is unable to find an online neighbor, making $k = 20$ the better choice even for the adaptive codec.

It is interesting to note that in the very early part of the simulation, the compressed push-pull with $k = 20$ performs *better* in the presence of churn than in its absence. This is because on this small timescale, node status is relatively stable, so the main effect of churn is the reduced set of online neighbors, approximating the effects of a smaller k, which helps the pivot codec.

6.5 Conclusions

In this chapter we presented several contributions: a novel push-pull averaging algorithm that supports compression of communication, its variant adapted to machine learning, and a novel codec (called pivot codec) for implementing compression. The push-pull algorithms can be used with any codec and the pivot codec can be used with any distributed algorithm that supports codecs.

The original features of the push-pull averaging algorithm include a mechanism to tolerate message drop failure, and a technique to support overlapping transactions with different neighbors. We also added a mechanism that we called flow compensation, which makes the stream of values over a given link smoother to improve compression. Another smoothing technique is a greediness parameter H that controls the magnitude of each transaction. We extended this algorithm with weighted average calculation, and made adjustments to adapt and optimize it for machine learning.

The pivot codec that we introduced is based on the intuition that in decentralized aggregation algorithms the values sent over a link are often correlated so compressing the stream is in fact similar to trying to guess a constant value on the other side of an overlay link.

We demonstrated experimentally that the novel codec is superior in the scenarios we studied in terms of the compression rate. We also demonstrated that the flow compensation mechanism indeed improves performance, although the pivot codec dominates the other codecs from related work even without the flow compensation mechanism. We saw that the push-pull protocol is highly robust to overlapping transactions as well, and in general outperforms the synchronized iteration algorithm independently of the codec used.

We also evaluated the codec-based push-pull learning algorithm, and found that the method is competitive in the scenarios we studied. We also obtained considerable extra compression with the help of transfer learning, where, instead of the 784 raw MNIST features, we used only 78 features (extracted by a Fashion-MNIST model) and the linear model over this compressed feature set still allowed us to outperform the linear model that used the original 784 raw features.

Contribution

In this chapter, the contributions of the author were: the design and evaluation of the compressed push-pull averaging algorithm and the pivot codec; and the design of the compressed push-pull learning algorithm.

Algorithm 6.1 Robust push-pull averaging

```
1: $x$ is the local approximation of the average, initially the local value to be averaged.
2: $u_{i,in}$ and $u_{i,out}$ record the number of times the local value was updated as a result of an incoming push or pull message from $i$, respectively.
3: $s_i$ is the value that was sent in the last push message to $i$.
4: $\delta_{i,out}, \delta_{i,in}$ are the last push, or pull transfers to $i$, respectively.
5: $id_i$ is the current unique ID created when sending the latest push message to $i$, initially 0.
6: $id_{max,i}$ is the maximal unique ID received in any push message from $i$, initially $-\infty$.
7:
8: procedure onNextCycle                                ▷ called every $\Delta$ time units
9:     $i \leftarrow$ randomOutNeighbor()
10:    $s_i \leftarrow x$
11:    $id_i \leftarrow id_i + 1$
12:    send push message $(u_{i,out}, s_i, id_i)$ to node $i$
13: end procedure
14:
15: procedure onPushMessage($u, s, id, i$)                ▷ received from node $i$
16:    if $id_{max,i} < id$ then
17:        $id_{max,i} \leftarrow id$
18:        if $u < u_{i,in}$ then        ▷ last pull has not arrived, roll back corresponding update
19:            $x \leftarrow x + \delta_{i,in}$
20:            $u_{i,in} \leftarrow u_{i,in} - 1$
21:        end if
22:        send pull message $(x, id)$ to node $i$
23:        update$(i, in, x, s)$
24:    end if
25: end procedure
26:
27: procedure onPullMessage($s, id, i$)                   ▷ received from node $i$
28:    if $id_i = id$ then
29:        update$(i, out, s_i, s)$
30:    end if
31: end procedure
32:
33: procedure update($i, d, s_{loc}, s_{rem}$)
34:    $u_{i,d} \leftarrow u_{i,d} + 1$
35:    $\delta_{i,d} \leftarrow H \cdot \frac{1}{2}(s_{loc} - s_{rem})$
36:    $x \leftarrow x - \delta_{i,d}$
37: end procedure
```

Algorithm 6.2 Compressed push-pull averaging with smoothing

```
1: $\xi_{i,in,loc}, \xi_{i,in,rem}, \xi_{i,out,loc}, \xi_{i,out,rem} \in \Xi$ are the states of the codecs for the local node
   and remote node $i$, initially $\xi_0$.
2: $f_{i,in}, f_{i,out}$ are the amounts of mass transferred so far to $i$, initially 0.
3: $\xi_{i,in',loc}, \xi_{i,in',rem}$, and $f_{i,in'}$ are the previous values of $\xi_{i,in,loc}, \xi_{i,in,rem}$, and $f_{i,in}$, ini-
   tially $\xi_0, \xi_0$, and 0, respectively.
4:
5: procedure onNextCycle                                ▷ called every $\Delta$ time units
6:     $i \leftarrow$ randomOutNeighbor()
7:     $s_i \leftarrow Q(\xi_{i,out,loc}, x + f_{i,out})$
8:     $id_i \leftarrow id_i + 1$
9:     send push message $(u_{i,out}, s_i, id_i)$ to node $i$
10: end procedure
11:
12: procedure onPushMessage$(u, s, id, i)$                 ▷ received from node $i$
13:     if $id_{max,i} < id$ then
14:         $id_{max,i} \leftarrow id$
15:         if $u < u_{i,in}$ then         ▷ last pull has not arrived, roll back corresponding
    update
16:             $x \leftarrow x + \delta_{i,in}$
17:             $u_{i,in} \leftarrow u_{i,in} - 1$
18:             $(\xi_{i,in,loc}, \xi_{i,in,rem}, f_{i,in}) \leftarrow (\xi_{i,in',loc}, \xi_{i,in',rem}, f_{i,in'})$
19:         end if
20:         $s_{pull} \leftarrow Q(\xi_{i,in,loc}, x + f_{i,in})$
21:         $(\xi_{i,in',loc}, \xi_{i,in',rem}, f_{i,in'}) \leftarrow (\xi_{i,in,loc}, \xi_{i,in,rem}, f_{i,in})$
22:         send pull message $(s_{pull}, id)$ to node $i$
23:         update$(i, in, s_{pull}, s)$
24:     end if
25: end procedure
26:
27: procedure onPullMessage$(s, id, i)$                    ▷ received from node $i$
28:     if $id_i = id$ then
29:         update$(i, out, s_i, s)$
30:     end if
31: end procedure
32:
33: procedure update$(i, d, s_{loc}, s_{rem})$
34:     $u_{i,d} \leftarrow u_{i,d} + 1$
35:     $\delta_{i,d} \leftarrow H \cdot \frac{1}{2}(K(\xi_{i,d,loc}, s_{loc}) - K(\xi_{i,d,rem}, s_{rem}) - 2f_{i,d})$
36:     $(\xi_{i,d,loc}, \xi_{i,d,rem}, f_{i,d}) \leftarrow (F(\xi_{i,d,loc}, s_{loc}), F(\xi_{i,d,rem}, s_{rem}), f_{i,d} + \delta_{i,d})$
37:     $x \leftarrow x - \delta_{i,d}$
38: end procedure
```

Algorithm 6.3 Compressed push-pull learning (Part 1)

```
1: $w$ is the local model.
2: $t$ is the age of the local model.
3: $D$ is the local data set.
4: $u_{i,in}$ and $u_{i,out}$ record the number of times the local model was updated as a result
   of an incoming push or pull message from $i$, respectively.
5: $s_i$ and $\widehat{s}_i$ are the encoded model and model age that were sent in the last push
   message to $i$.
6: $\delta_{i,out}, \delta_{i,in}$ are the last push, or pull parameter transfers to $i$, respectively.
7: $\widehat{\delta}_{i,out}, \widehat{\delta}_{i,in}$ are the last push, or pull age transfers to $i$, respectively.
8: $id_i$ is the current unique ID created when sending the latest push message to $i$,
   initially 0.
9: $id_{max,i}$ is the maximal unique ID received in any push message from $i$, initially
   $-\infty$.
10: $\xi_{i,in,loc}, \xi_{i,in,rem}, \xi_{i,out,loc}, \xi_{i,out,rem} \in \Xi$ are the states of the codecs for the local node
    and remote node $i$, with initial values of $\xi_0$.
11: $\xi_{i,in',loc}$ and $\xi_{i,in',rem}$ are the previous values of $\xi_{i,in,loc}$ and $\xi_{i,in,rem}$, with initial
    values of $\xi_0$.
12:
13: procedure onNextCycle                                    ▷ Called every $\Delta$ time units
14:     $(w, t) \leftarrow$ train$(w, t, D)$
15:     $i \leftarrow$ randomOutNeighbor()
16:     $s_i \leftarrow Q(\xi_{i,out,loc}, w)$                       ▷ Model encoded and saved
17:     $\widehat{s}_i \leftarrow t$
18:     $id_i \leftarrow id_i + 1$
19:     send push message $(u_{i,out}, s_i, \widehat{s}_i, id_i)$ to node $i$
20: end procedure
```

Algorithm 6.4 Compressed push-pull learning (Part 2)

```
21: procedure ONPUSHMESSAGE($u, s, \widehat{s}, id, i$)    ▷ Received from node $i$
22:     if $id_{max,i} < id$ then    ▷ This is not an old, out-of-order message
23:         $id_{max,i} \leftarrow id$
24:         if $u < u_{i,in}$ then ▷ Last pull has not arrived, reverse corresponding update
25:             $w \leftarrow (t \cdot w + \delta_{i,in})/(t + \widehat{\delta}_{i,in})$
26:             $t \leftarrow t + \widehat{\delta}_{i,in}$
27:             $u_{i,in} \leftarrow u_{i,in} - 1$
28:             $(\xi_{i,in,loc}, \xi_{i,in,rem}) \leftarrow (\xi_{i,in',loc}, \xi_{i,in',rem})$    ▷ Previous codec states are restored
29:         end if
30:         $s_{pull} \leftarrow Q(\xi_{i,in,loc}, w)$
31:         $(\xi_{i,in',loc}, \xi_{i,in',rem}) \leftarrow (\xi_{i,in,loc}, \xi_{i,in,rem})$  ▷ Codec states are backed up before update
32:         send pull message $(s_{pull}, t, id)$ to node $i$
33:         update$(i, in, s_{pull}, t, s, \widehat{s})$
34:     end if
35: end procedure
36:
37: procedure ONPULLMESSAGE($s, \widehat{s}, id, i$)    ▷ Received from node $i$
38:     if $id_i = id$ then ▷ This is the answer for the last push message, not an old one
39:         update$(i, out, s_i, \widehat{s}_i, s, \widehat{s})$    ▷ The node uses the same data it sent, not the current local model
40:     end if
41: end procedure
42:
43: procedure UPDATE($i, d, s_{loc}, \widehat{s}_{loc}, s_{rem}, \widehat{s}_{rem}$)
44:     $u_{i,d} \leftarrow u_{i,d} + 1$
45:     $\delta_{i,d} \leftarrow H \cdot \frac{1}{2}(\widehat{s}_{loc} \cdot K(\xi_{i,d,loc}, s_{loc}) - \widehat{s}_{rem} \cdot K(\xi_{i,d,rem}, s_{rem}))$ ▷ Models are decoded and weighted by age
46:     $\widehat{\delta}_{i,d} \leftarrow H \cdot \frac{1}{2}(\widehat{s}_{loc} - \widehat{s}_{rem})$
47:     $(\xi_{i,d,loc}, \xi_{i,d,rem}) \leftarrow (F(\xi_{i,d,loc}, s_{loc}), F(\xi_{i,d,rem}, s_{rem}))$  ▷ Codec states are updated
48:     $w \leftarrow (t \cdot w - \delta_{i,d})/(t - \widehat{\delta}_{i,d})$    ▷ The updates operate on $tw$, not $w$, hence the conversions
49:     $t \leftarrow t - \widehat{\delta}_{i,d}$    ▷ Notice that this new $t$ is used above
50: end procedure
```

Bibliography

[1] Waseem Ahmad and Ashfaq Khokhar. Secure aggregation in large scale overlay networks. In *IEEE Global Telecommunications Conference (GLOBECOM ’06)*, 2006.

[2] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In *Esann*, volume 3, page 3, 2013.

[3] Jimmy Ba and Diederik Kingma. Adam: A method for stochastic optimization. In *3rd Intl. Conf. on Learning Representations (ICLR)*, 2015.

[4] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[5] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), August 2019.

[6] Árpád Berta, Vilmos Bilicki, and Márk Jelasity. Defining and understanding smartphone churn over the internet: a measurement study. In *Proceedings of the 14th IEEE International Conference on Peer-to-Peer Computing (P2P 2014)*. IEEE, 2014.

[7] Danny Bickson, Tzachy Reinman, Danny Dolev, and Benny Pinkas. Peer-to-peer secure multi-party numerical computation facing malicious adversaries. *Peer-to-Peer Networking and Applications*, 3(2):129–144, 2010.

[8] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[10] Michael Blot, David Picard, Nicolas Thome, and Matthieu Cord. Distributed optimization for deep learning with gossip exchange. *Neurocomputing*, 330:287–296, 2019.

[11] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[12] Léon Bottou. Stochastic gradient descent tricks. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700 of *Lecture Notes in Computer Science*, pages 421–436. Springer Berlin Heidelberg, 2012.

[13] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[14] R. W. Brockett and D. Liberzon. Quantized feedback stabilization of linear systems. *IEEE Transactions on Automatic Control*, 45(7):1279–1289, 2000.

[15] S. Buckingham Shum, K. Aberer, A. Schmidt, S. Bishop, P. Lukowicz, S. Anderson, Y. Charalabidis, J. Domingue, S. Freitas, I. Dunwell, B. Edmonds, F. Grey, M. Haklay, M. Jelasity, A. Karpištšenko, J. Kohlhammer, J. Lewis, J. Pitt, R. Sumner, and D. Helbing. Towards a global participatory platform. *The European Physical Journal Special Topics*, 214(1):109–152, 2012.

[16] Ruggero Carli, Francesco Bullo, and Sandro Zampieri. Quantized average consensus via dynamic coding/decoding schemes. *Intl. Journal of Robust and Nonlinear Control*, 20(2):156–175, 2010.

[17] Ruggero Carli, Fabio Fagnani, Paolo Frasca, and Sandro Zampieri. Gossip consensus algorithms via quantized communication. *Automatica*, 46(1):70–80, 2010.

[18] Chris Clifton, Murat Kantarcioglu, Jaideep Vaidya, Xiaodong Lin, and Michael Y. Zhu. Tools for privacy preserving distributed data mining. *SIGKDD Explor. Newsl.*, 4(2):28–34, December 2002.

[19] Gábor Danner, Árpád Berta, István Hegedűs, and Márk Jelasity. Robust fully distributed mini-batch gradient descent with privacy preservation. *Security and Communication Networks*, 2018:6728020, 2018.

[20] Gábor Danner, István Hegedűs, and Márk Jelasity. Decentralized machine learning using compressed push-pull averaging. In *Proceedings of the 1st International Workshop on Distributed Infrastructure for Common Good*, pages 31–36, 2020.

[21] Gábor Danner and Márk Jelasity. Fully distributed privacy preserving mini-batch gradient descent learning. In Alysson Bessani and Sara Bouchenak, editors, *Proceedings of the 15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2015)*, volume 9038 of *Lecture Notes in Computer Science*, pages 30–44. Springer International Publishing, 2015.

[22] Gábor Danner and Márk Jelasity. Robust decentralized mean estimation with limited communication. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018*, volume 11014 of *Lecture Notes in Computer Science*, pages 447–461. Springer International Publishing, 2018.

[23] Gábor Danner and Márk Jelasity. Token account algorithms: The best of the proactive and reactive worlds. In *Proceedings of The 38th International Conference on Distributed Computing Systems (ICDCS 2018)*, pages 885–895. IEEE Computer Society, 2018.

[24] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc.

[25] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *J. Mach. Learn. Res.*, 13(1):165–202, January 2012.

[26] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[28] Dheeru Dua and Casey Graff. UCI machine learning repository, 2019.

[29] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 27:1–27:12, New York, NY, USA, 2007. ACM.

[30] Cynthia Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54(1):86–95, January 2011.

[31] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 486–503. Springer Berlin Heidelberg, 2006.

[32] European Commission. General data protection regulation (GDPR), 2018.

[33] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Maxime Monod. Boosting gossip for live streaming. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10. IEEE, August 2010.

[34] M. Fu and L. Xie. Finite-level quantized feedback control for linear systems. *IEEE Transactions on Automatic Control*, 54(5):1165–1170, 2009.

[35] Lodovico Giaretta and Šarūnas Girdzijauskas. Gossip learning: Off the beaten path. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1117–1124, December 2019.

[36] Kevin Gimpel, Dipanjan Das, and Noah A. Smith. Distributed asynchronous online learning for natural language processing. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL'10)*, pages 213–222, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[37] Shuguo Han, Wee Keong Ng, Li Wan, and Vincent C. S. Lee. Privacy-preserving gradient-descent methods. *IEEE Transactions on Knowledge and Data Engineering*, 22(6):884–899, 2010.

[38] István Hegedűs, Árpád Berta, Levente Kocsis, András A. Benczúr, and Márk Jelasity. Robust decentralized low-rank matrix decomposition. *ACM Transactions on Intelligent Systems and Technology*, 7(4):62:1–62:24, May 2016.

[39] István Hegedűs, Gábor Danner, and Márk Jelasity. Decentralized recommendation based on matrix factorization: A comparison of gossip and federated learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 317–332. Springer, 2019.

[40] István Hegedűs, Gábor Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In José Pereira and Laura Ricci, editors, *Proceedings of the 19th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2019)*, volume 11534 of *Lecture Notes in Computer Science*, pages 74–90. Springer International Publishing, 2019.

[41] István Hegedűs, Gábor Danner, and Márk Jelasity. Decentralized learning works: An empirical comparison of gossip learning and federated learning. *Journal of Parallel and Distributed Computing*, 148:109–124, 2021.

[42] Chenghao Hu, Jingyan Jiang, and Zhi Wang. Decentralized federated learning: A segmented gossip approach. In *The 1st International Workshop on Federated Machine Learning for User Privacy and Data Confidentiality (IJCAI Workshop)*, 2019.

[43] Mohsan Jameel, Josif Grabocka, Mofassir ul Islam Arif, and Lars Schmidt-Thieme. Ring-star: A sparse topology for faster model averaging in decentralized parallel SGD. In *Decentralized Machine Learning at the Edge (ECML PKDD 2019 Workshop)*, 2019.

[44] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[45] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[46] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. Secure peer sampling. *Computer Networks*, 54(12):2086–2098, 2010.

[47] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Fault-tolerant aggregation for dynamic networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 37–43, 2010.

[48] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proc. 44th Annual IEEE Symp. on Foundations of Comp. Sci. (FOCS'03)*, pages 482–491. IEEE Computer Society, 2003.

[49] Anastasia Koloskova, Sebastian Stich, and Martin Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3478–3487, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[50] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *Private Multi-Party Machine Learning (NIPS 2016 Workshop)*, 2016.

[51] Anusha Lalitha, Shubhanshu Shekhar, Tara Javidi, and Farinaz Koushanfar. Fully decentralized federated learning. In *Bayesian Deep Learning (NIPS 2018 Workshop)*, 2018.

[52] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, November 1998.

[53] T. Li, M. Fu, L. Xie, and J. F. Zhang. Distributed consensus with limited communication data rate. *IEEE Transactions on Automatic Control*, 56(2):279–292, 2011.

[54] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5330–5340. Curran Associates, Inc., 2017.

[55] M. Lichman. UCI machine learning repository, 2013.

[56] Boris Lubachevsky and Debasis Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit radius. *Journal of the ACM*, 33(1):130–150, January 1986.

[57] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[58] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Jerry Zhu, editors, *Proceedings*

of the 20th International Conference on Artificial Intelligence and Statistics, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[59] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Proc. 9th IEEE Intl. Conf. Peer-to-Peer Computing (P2P 2009)*, pages 99–100, Seattle, Washington, USA, September 2009. IEEE. extended abstract.

[60] G. N. Nair, F. Fagnani, S. Zampieri, and R. J. Evans. Feedback control under data rate constraints: An overview. *Proc. IEEE*, 95(1):108–137, 2007.

[61] Juan A. M. Naranjo, Leocadio G. Casado, and Márk Jelasity. Asynchronous privacy-preserving iterative computation on peer-to-peer networks. *Computing*, 94(8–10):763–782, 2012.

[62] Gerhard Niederbrucker and Wilfried N. Gansterer. Robust gossip-based aggregation: A practical point of view. In *Proc. Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 133–147, 2013.

[63] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and cooperation in networked multi-agent systems. *Proc. IEEE*, 95(1):215–233, 2007.

[64] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.

[65] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.

[66] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 1999.

[67] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 337–348, New York, NY, USA, 2007. ACM.

[68] Arun Rajkumar and Shivani Agarwal. A differentially private stochastic gradient descent algorithm for multiparty classification. *JMLR Workshop and Conference Proceedings*, 22:933–941, 2012. Proceedings of AISTATS'12.

[69] P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conf. (DCC'06)*, pages 133–142, 2006.

[70] Luis Rodrigues, Sidath Handurukande, José Pereira, Rachid Guerraoui, and Anne-Marie Kermarrec. Adaptive gossip-based broadcast. In *International Conference on Dependable Systems and Networks (DSN-2003)*, pages 47–56, June 2003.

[71] Roberto Roverso, Jim Dowling, and Márk Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing (P2P 2013)*. IEEE, 2013.

[72] Jared Saia and Mahdi Zamani. Recent results in scalable multi-party computation. In *41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'15)*, volume 8939 of *LNCS*. Springer, 2015.

[73] S. Savazzi, M. Nicoli, and V. Rampa. Federated learning with cooperating devices: A consensus approach for massive iot networks. *IEEE Internet of Things Journal*, 2020.

[74] Ali Sayed. Adaptation, learning, and optimization over networks. *Found. Trends Mach. Learn.*, 7(4-5):311–801, July 2014.

[75] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming B*, 2010.

[76] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE Transactions on Medical Imaging*, 35(5):1285–1298, 2016.

[77] Emil Sit, Andreas Haeberlen, Frank Dabek, Byung-Gon Chun, Hakim Weatherspoon, Robert Morris, M Frans Kaashoek, and John Kubiatowicz. Proactive replication for data durability. In *The 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.

[78] Daniel Stutzbach, Reza Rejaie, Nick Duffield, Subhabrata Sen, and Walter Willinger. On unbiased sampling for unstructured peer-to-peer networks. *IEEE/ACM Transactions on Networking*, 17(2):377–390, April 2009.

[79] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In *Proc. 34th Intl. Conf. Machine Learning, (ICML)*, pages 3329–3337, 2017.

[80] Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. D^2: Decentralized training over decentralized data. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4848–4856, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[81] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[82] Ji Wang, Bokai Cao, Philip S. Yu, Lichao Sun, Weidong Bao, and Xiaomin Zhu. Deep learning towards mobile applications. In *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1385–1393, July 2018.

[83] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[84] Ran Wolff, Kanishka Bhaduri, and Hillol Kargupta. Local l2-thresholding based data mining in peer-to-peer systems. In *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*, pages 430–441. SIAM, 2006.

[85] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[86] Lin Xiao and Stephen Boyd. Fast linear iterations for distributed averaging. *Systems & Control Letters*, 53(1):65–78, 2004.

[87] Lin Xiao, Stephen Boyd, and Sanjay Lall. A scheme for robust distributed sensor fusion based on average consensus. In *IPSN'05: Proc. 4th Intl. Symp. on Inf. Proc. in Sensor Networks*, page 9, 2005.

[88] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

[89] Yuchen Zhang, John C. Duchi, and Martin J. Wainwright. Communication-efficient algorithms for statistical optimization. *J. Mach. Learn. Res.*, 14(1):3321–3363, January 2013.

[90] M. Zhu and S. Martinez. On the convergence time of asynchronous distributed quantized averaging algorithms. *IEEE Transactions on Automatic Control*, 56(2):386–390, 2011.

Summary

Gossip learning is a fully distributed machine learning framework, where nodes communicate directly, exchanging their models. In this thesis we introduced a number of techniques to improve gossip learning. Here, we give a summary of each of the four major parts of the dissertation.

Gossip Learning with Privacy Preservation

In gossip learning, colluding nodes may obtain information about the training example of another node if they see the model right before and after it is updated by the node.

In Chapter 3, we proposed a secure mini-batch gradient method to improve the privacy of the users. We assumed a semi-honest adversary: the corrupted nodes still follow the protocol but the adversary can see the internal state of such nodes. Our solution is to replace the local update step with a distributed mini-batch approach. For each step of the random walk, when a node receives a model to update, it coordinates the distributed computation of a mini-batch gradient and then uses this gradient to update the model.

The first step for computing a mini-batch gradient is to create a temporary group of random nodes that form the mini-batch; we do this by building a rooted overlay tree. We proposed the building of a binomial tree for optimal performance, with a "trunk" that is needed for improved security.

Then, we can use our novel secure sum algorithm to aggregate the gradients of the participants. It builds upon a secret sharing scheme where a secret value is split into multiple shares such that all the shares are needed to obtain any information. The basic idea of the algorithm is to divide the local value at each node into shares, encrypt these with asymmetric additively homomorphic encryption (e.g. via the Paillier cryptosystem), and send them to the root via the chain of ancestors. Although the shares travel together, they are encrypted with the public keys of different ancestors. Along the route, the arrays of shares are aggregated, and periodically re-encrypted. Finally, the root calculates the sum.

We can achieve high levels of robustness and good scalability by exploiting the fact that the mini-batch gradient algorithm does not require the sum to be precise. The algorithm is designed to calculate a partial sum in the event of node failures.

We evaluated the protocol in realistic simulations using a smartphone trace to simulate churn. We demonstrated on a number of learning tasks that the approach is indeed practically viable.

The contributions of the author are:

- A scalable and robust secure sum protocol that is able to securely compute a partial sum even in the event of failures and limited collusion of nodes;
- A proof about its capability of preventing the collusion attack;
- A decentralized mini-batch gradient descent method based on the building of a k-trunked binomial overlay tree and the above protocol.

Comparison of Federated and Gossip Learning

In federated learning, the workers perform machine learning over their own data and the master merely aggregates the resulting models without seeing any raw data, not unlike the parameter server approach. Gossip learning is a decentralized alternative to federated learning that does not require an aggregation server or any central component. The natural hypothesis is that gossip learning is strictly less efficient than federated learning due to it relying on a more basic infrastructure: only message passing and no cloud resources.

In Chapter 4, we questioned this hypothesis. We presented a thorough comparison of the two approaches. The experimental scenarios included a real churn trace collected over mobile phones, different network sizes and different distributions of the training data over the devices. Also, we applied subsampling to reduce communication in both approaches; that is, we sent only random subsets of the model parameters. Here, we introduced a new subsampling technique for gossip learning based on partitioned models where each partition has its own age parameter. Instead of sampling parameters independently, one of the partitions is chosen. This way, during model merging, the model parameters can be averaged with appropriate weights without increasing communication costs.

We compared federated and gossip learning in terms of convergence time and model quality, assuming that both approaches utilize the same amount of communication resources in the same scenarios. We also performed a systematic hyperparameter analysis. Surprisingly, the best gossip variants performed comparably with the best federated learning variants overall, thus providing a fully decentralized alternative to federated learning.

The contributions of the author are:

- The partition-based sampling technique;
- The design and development of churn-related modules of the simulator;
- Participation in the design of the improved aggregation algorithm for federated learning;
- Participation in the planning of experiments;
- The optimization of hyperparameters.

Gossip Learning with Adaptive Flow Control

Many decentralized algorithms allow both proactive and reactive implementations. Examples include gossip protocols for broadcasting and decentralized computing, as well as chaotic matrix iteration algorithms. In proactive systems, nodes communicate at a fixed rate in regular intervals, while in reactive systems they communicate in response to certain events such as the arrival of fresh data. Although reactive algorithms tend to stabilize/converge/self-heal much faster, they have serious drawbacks: they may overload the network, and they may also cause starvation when the number of messages circulating in the system becomes too low. Proactive algorithms do not have these problems, but nodes waste a lot of time sitting on fresh information.

In Chapter 5, we proposed the token account framework, a novel family of adaptive protocols that apply rate limiting inspired by the token bucket algorithm to prevent uncontrolled bursts, but they also include proactive communication to prevent starvation. With the help of our traffic shaping service, some applications approach the speed of the reactive implementation, while maintaining strong guarantees regarding the total communication cost and burstiness. In a nutshell, these algorithms grant a token to each node in regular periods, and sending a message costs a token. A token can be spent immediately (proactive operation), or later, when a message is received (reactive operation). The more tokens a node has, the more eager it is to spend them, possibly sending multiple reactive messages at once. When there are too few messages circulating, the token accounts start to fill up, encouraging an increase in network activity. We performed simulation experiments in different scenarios including a real smartphone availability trace. Our results suggest up to a fourfold speedup in a broadcast application, and an order of magnitude speedup in the case of gossip learning, when compared to the purely proactive implementation.

To evaluate this token-based flow control technique with the mergeable version of gossip learning, we performed machine learning over three different datasets. We also introduced a partitioned variant of the token account algorithm, to properly make use of sampling-based compression. Here, each partition has its own token account. Our results confirmed that the token-based flow control approach outperforms proactive gossip learning. Furthermore, it can achieve a performance comparable to federated learning when the distribution of training examples is unbiased. However, to achieve these results, the compression mechanism must be based on partitioning, as opposed to simple subsampling. The reason is that this way, the different partitions can form "hot potato" chains separately, whereas with subsampling, these chains cannot form because sampling picks different weights for each step of the random walk.

The contributions of the author are:

- The design and evaluation of the token account algorithm;
- An analytical derivation of the average number of tokens in the system;
- The design of the partitioned token gossip learning algorithm.

Gossip Learning Using Compressed Averaging

Mean estimation, also known as average consensus, is an important computational primitive in decentralized systems. When the average of large vectors has to be computed, as in distributed data mining applications, reducing the communication cost becomes a key design goal. One way of reducing the communication cost is to add dynamic stateful encoder-decoder pairs (codecs) to traditional mean estimation protocols. In this approach, each element of a vector message is encoded in a few bits and decoded by the recipient node. However, due to this encoding and decoding mechanism, these protocols are much more sensitive to benign failure such as message drop and message delay. Properties such as mass conservation are harder to guarantee. Hence, known approaches are formulated under strong assumptions such as reliable communication, atomic non-overlapping transactions or even full synchrony.

In Chapter 6, we proposed a communication efficient algorithm that supports codecs even if transactions overlap and the nodes are not synchronized. The algorithm is based on push-pull averaging, with novel features to support fault tolerance and compression. With the help of simple counters, it is able to detect whether the transferred amount (and the codec state) became inconsistent across the link due to message loss, and rolls back the state to a consistent one. As an independent contribution, we also proposed a novel adaptive codec, called the pivot codec. We demonstrated experimentally that our algorithm improves the performance of existing codecs and the novel pivot codec dominates the competing codecs in the scenarios we studied.

Furthermore, we proposed a novel variant of gossip learning that uses this codec-based compression to achieve a higher communication efficiency than previous methods could based on subsampling. The algorithm periodically trains the local model and performs the weighted averaging of the models in the network. Among our machine learning experiments we also included a transfer learning scenario. This means that we trained a relatively small model on top of a high quality pre-trained feature set that is fixed.

The contributions of the author are:

- The design and evaluation of the compressed push-pull averaging algorithm;
- The design and evaluation of the pivot codec;
- The design of the compressed push-pull learning algorithm.

Összefoglalás

A pletyka alapú tanulás egy teljesen elosztott gépi tanulási keretrendszer, ahol a hálózatra kötött eszközök (csomópontok) központi szerver használata nélkül, közvetlenül egymással kommunikálnak, továbbítva egymásnak gépi tanuló modelljeiket. Ezen disszertációban számos új módszert mutattunk be, melyek hatékonyabbá vagy biztonságosabbá teszik a pletyka tanulást.

Pletyka alapú tanulás adatvédelemmel

A pletyka alapú tanulás során a hálózatban véletlen sétákat tesznek meg a modellek, és minden lépésben a helyi adatokon tanítjuk őket. Egymással összejátszó csomópontok adatokat szerezhetnek meg egy másik csomópontról, ha az ott végzett tanítás előtti és utáni modellváltozat is a birtokukban van.

A 3. fejezetben javaslunk egy biztonságos mini-batch gradiens módszert a felhasználók adatvédelmének elősegítése érdekében. Az általunk használt ellenfélmodellben feltesszük, hogy az összejátszó csomópontok csak megfigyelnek, azaz nem módosítják az algoritmus működését. A módszerünkben a véletlen séta minden lépésében elvégzünk egy elosztott mini-batch számítást, és az összegzett gradiensek alapján végezzük el a tanítást.

Ezen számításhoz először létre kell hozni egy véletlen csomópontokból álló ideiglenes csoportot. Ehhez a kezdeményező csomópontból, mint gyökérből kiindulva egy fa topológiájú fedőhálót építünk. Egy "törzzsel" rendelkező binomiális fa építését javasoljuk a hatékonyság és biztonság elérése érdekében.

Ezután az új biztonságos összegző algoritmusunkkal ki tudjuk számolni a gradiensek összegét. Ehhez felhasználunk egy titokfelosztási módszert, amellyel egy titkos számértéket úgy tudunk több részre bontani, hogy csak az összes rész birtokában lehessen információhoz jutni az eredeti értékről. Az algoritmusunk alapötlete az, hogy minden csomópont a gradiensét ezzel a módszerrel több részre bontja, ezeket egy asszimetrikus additívan homomorfikus titkosítási rendszerrel betitkosítja, és elküldi a gyökérbe a szülők láncolatán keresztül. Ezek a részek együtt haladnak a hálózatban, de különböző ősök publikus kulcsával vannak titkosítva. Az út során a különböző gradiensekből származó részeket aggregáljuk és újrakódoljuk. Végül a gyökér kiszámítja az összeget.

A mini-batch gradiens algoritmusnak nincs szüksége pontos összegre, és ezt kihasználva magas hibatűrést és skálázódást tudunk elérni. Ha egyes csomópontok leszakadnak, az algoritmus egy részösszeget számít ki.

Kiértékeltük az algoritmust realisztikus szimulációkban, okostelefonok hálózati elérhetőségének méréséből származó idősorokat is felhasználva. Több gépi tanulási feladaton is megmutattuk, hogy a módszer megvalósítható.

A szerző hozzájárulásai:

- egy skálázható és robusztus biztonságos összegző protokoll, amely hibák és bizonyos fokú összejátszás esetén is képes részösszeg biztonságos kiszámítására;
- ezen protokollról szóló bizonyítás;
- egy k hosszú törzzsel rendelkező binomiális fa építésén, és a fenti összegző protokollon alapuló decentralizált mini-batch módszer.

A federated learning és a pletyka tanulás összehasonlítása

A federated learning egy mester-szolga architektúrát használ: a szolgák (csomópontok) gépi tanulást végeznek a saját adatukon, a mester (szerver) pedig a tőlük kapott modelleket kiátlagolja és visszaküldi. A pletyka alapú tanulás egy decentralizált alternatívát kínál, mivel nem igényel szervert vagy egyéb központi komponenst. Adódik a természetes feltevés, hogy a pletyka tanulás szigorúan kevésbé hatékony, mint a federated learning, mivel nem vesz igénybe felhő erőforrásokat.

A 4. fejezetben megkérdőjeleztük ezt a feltevést. Bemutattuk a két megközelítés alapos összehasonlítását. Szimulációs kísérletinkben megvizsgáltunk különböző méretű hálózatokat és a tanítópéldák különböző megoszlását a csomópontok között, továbbá okostelefonok hálózati elérhetőségének méréséből származó idősorokat is alkalmaztunk. Mindkét megközelítésben paraméter-mintavételezést is alkalmaztunk az adatforgalom csökkentésére; vagyis a modell paramétereinek csak véletlenszerű részhalmazait küldtük el. Egy új, particionált modelleken alapuló mintavételezési technikát vezettünk be a pletyka tanulás számára, ahol minden partíció saját életkor paraméterrel rendelkezik. A paraméterek független mintavételezése helyett az egyik partíció kerül kiválasztásra. Így a modellösszeolvasztás során a modell paraméterei a megfelelő súlyokkal átlagolhatók a kommunikációs költségek növekedése nélkül.

Összehasonlítottuk a federated és a pletyka tanulást a konvergenciaidő és a modell minősége szempontjából, feltételezve, hogy mindkét megközelítés ugyanannyi kommunikációs erőforrást használhat ugyanabban a forgatókönyvben. Továbbá szisztematikus hiperparaméter elemzést is végeztünk. Meglepő módon a pletyka tanulás legjobb változatai összességében összehasonlíthatóan teljesítettek a federated learning legjobb variánsaival, ezáltal egy teljesen decentralizált alternatívát biztosítva.

A szerző hozzájárulásai:

- a paraméter-mintavételezéses tömörítés egy új, partíciói-alapú változata;
- új churn-kezelő peersim modulok tervezése és implementálása;
- részvétel a federated learning-ben használható paraméter-szelekciós aggregáció kidolgozásában;
- részvétel a kísérletek megtervezésében;
- a meta-paraméterek optimalizálása.

Pletyka alapú tanulás adaptív áramlásvezérléssel

Sok decentralizált algoritmus esetén lehetőség van mind a proaktív, mind a reaktív megvalósításra. A példák közé tartoznak decentralizált számítás és broadcast pletykaprotokollok, valamint kaotikus mátrix iterációs algoritmusok. A proaktív rendszerekben a csomópontok rögzített ütemben, rendszeres időközönként kommunikálnak, míg a reaktív rendszerek bizonyos eseményekre, például friss adatok érkezésére reagálva teszik ezt. Bár a reaktív algoritmusok jellemzően sokkal gyorsabban stabilizálódnak/konvergálnak/öngyógyulnak, mégis komoly hátrányai vannak: egyrészt túlterhelhetik a hálózatot, másrészt a rendszerben keringő üzenetek száma túl alacsonyra is csökkenhet. A proaktív algoritmusoknak nincsenek ilyen problémái, de a csomópontok sok időt elpazarolnak friss információkon ülve.

Az 5. fejezetben bemutattuk a token számla keretrendszert, olyan adaptív protokollok egy új családját, amelyek forgalomkorlátozást alkalmaznak a kontrollálatlan forgalomgenerálás megakadályozására, de az üzenetek kihalásának megelőzésére irányuló proaktív kommunikációt is végeznek. Forgalomformáló szolgáltatásunk segítségével egyes alkalmazások megközelítik a reaktív implementáció sebességét, miközben szilárd garanciákat tartunk fenn a kommunikáció összköltségére vonatkozóan. Dióhéjban, ezek az algoritmusok minden csomópontnak adnak egy tokent rendszeres időközönként, és egy üzenet elküldése egy tokenbe kerül. A token annak megkapáskor azonnal elkölthető (proaktív üzenetküldés), vagy később, amikor üzenet érkezik (reaktív üzenetküldés). Minél több tokennel rendelkezik egy csomópont, annál kevésbé spórol velük, akár több reaktív üzenetet is küldhet egyszerre. Ha túl kevés üzenet kering, a token számlák elkezdenek megtelni, ami a hálózati aktivitás növelését ösztönzi. Szimulációs kísérleteket végeztünk különböző forgatókönyvekben, valódi okostelefon-elérhetőségi idősorokat is felhasználva. Eredményeink akár négyszeres gyorsulást is mutatnak egy broadcast alkalmazásban, és nagyságrendi gyorsulást pletykatanulás esetén a tisztán proaktív megvalósításhoz képest.

Hogy kiértékeljük ezen token-alapú áramlásvezérlési technikát a pletykatanulás modell-összeolvasztást is alkalmazó változatával, gépi tanulást végeztünk három különböző adatbázison. Bevezettük a token számla algoritmus egy particionált változatát is, hogy ki tudja használni a mintavételezésen alapuló tömörítés előnyeit. Ebben a változatban minden partíciónak saját token számlája van. Eredményeink megerősítették, hogy a token-alapú áramlásvezérlésen alapuló megközelítés felülmúlja a proaktív pletykatanulást. Ezenkívül a federated learning teljesítményével összehasonlítható teljesítményt is elérhet, ha a tanítópéldák címke szerinti eloszlása egyenletes. Azonban ezen eredmények eléréséhez a tömörítési mechanizmusnak particionáláson kell alapulnia. Ennek az az oka, hogy így az egyes partíciók önálló "forró krumpli" üzenetláncokat generálhatnak, míg az egyszerű mintavételezéses tömörítés esetén ezek a láncok nem tudnak kialakulni, mert az különböző paramétereket választ a véletlen séta minden lépésében.

A szerző hozzájárulásai:

- a fedőhálózaton történő véletlen séták számát dinamikus egyensúlyban tartó (a hálózati csomagok forgalmának limitálására használt token bucket algoritmust általánosító) algoritmus és kiértékelése;
- a token-szám eloszlás elméleti vizsgálata;
- az algoritmus alkalmazása gépi tanulásra, kombinálása paraméter-mintavételezéses tömörítéssel.

Tömörített átlagoláson alapuló pletyka tanulás

A decentralizált átlagszámítás fontos számítási primitív decentralizált rendszerekben. Amikor a nagy vektorok átlagát kell kiszámítani, mint például elosztott adatbányászati alkalmazások esetén, a kommunikációs költségek csökkentése kulcsfontosságú tervezési céllá válik. Ennek egyik módja az, hogy dinamikus kódoló-dekódoló párokkal (kodekekkel) bővítjük a hagyományos átlagszámító protokollokat. Ebben a megközelítésben az üzenet (vektor) minden elemét néhány bitben kódolja a küldő csomópont, amit dekódol a fogadó csomópont. Ennek a kódolási és dekódolási mechanizmusnak köszönhetően azonban ezek a protokollok sokkal érzékenyebbek a hibákra, például az üzenet elvesztésére vagy késleltetésére. Az olyan tulajdonságokat, mint a tömegmegmaradás, nehezebb garantálni. Ezért az ismert megközelítéseket olyan erős feltevéseket használnak, mint a megbízható kommunikáció, atomi, nem-átfedő tranzakciók vagy akár teljes szinkronizáció.

A 6. fejezetben javasoltunk egy kommunikációban hatékony algoritmust, amely akkor is támogatja a kodekeket, ha a tranzakciók átfedhetnek, és a csomópontok nincsenek szinkronizálva. Az algoritmus push-pull átlagoláson alapul, új hibatűrést és a tömörítést támogató megoldásokkal kiegészítve. Egyszerű számlálók segítségével képes észlelni, hogy az átküldött össztömeg (és a kodek állapota) üzenetvesztés miatt inkonzisztenssé vált-e a két végpont között, és visszagörgeti az állapotot egy konzisztensre. Ezenfelül egy új adaptív kodeket is javasoltunk, a pivot kodeket. Kísérletileg megmutattuk, hogy az algoritmusunk javítja a meglévő kodekek teljesítményét, az új pivot kodek felülmúlta a versengő kodekeket a vizsgálatainkban.

Ezenkívül a pletykatanulás egy új változatát javasoltuk, amely ezt a kodek alapú tömörítést használja, hogy nagyobb kommunikációs hatékonyságot érjen el, mint a korábbi, mintavételezésen alapuló módszerek. Az algoritmus adott időközönként tanítja a helyi modellt, és eközben a modellek súlyozott átlagolását végzi a hálózatban. Gépi tanulási kísérleteink során transzfertanulást is alkalmaztunk. Ez azt jelenti, hogy egy viszonylag kis modellt tanítottunk egy rögzített, jó minőségű, előre betanított neuronhálóból nyert jellemzőkészlet fölött.

A szerző hozzájárulásai:

- egy csökkentett kommunikációs igényű, robusztus decentralizált átlagoló algoritmus;
- a pivot kodek;
- az előbbieken alapuló gossip learning algoritmus.