Convex Optimization using Sparsified Stochastic Gradient Descent with Memory

Faculté Informatique et Communications
École Polytechnique Fédérale de Lausanne
Thèse de Master de

Jean-Baptiste Cordonnier

Supervisée par
Prof. Martin Jaggi, Machine Learning and Optimization, EPFL
Dr. Sebastian U. Stich, Machine Learning and Optimization, EPFL

Lausanne, EPFL, 2018
Abstract

The interest for distributed stochastic optimization has raised to train complex Machine Learning models with more data on distributed systems. Increasing the computation power speeds up the training but it faces a communication bottleneck between workers which hurts the scale-up of these distributed algorithms. Previous work tried to address this issue through quantization by broadcasting low precision updates (Seide et al., 2014; Alistarh et al., 2017) or through sparsification by sharing only the most important coordinates to update (Aji and Heafield, 2017; Lin et al., 2018). Even though the sparsification method works well in practice, it lacked a theoretical proof until now.

We propose a sparsification scheme for SGD where only a small constant number of coordinates are applied at each iteration. The amount of data to be communicated is drastically reduced while the $O(1/T)$ convergence rate of vanilla SGD is preserved. Our concise proof extends to parallel setting and gains a linear speed up in the number of workers. We experiment with sparsified SGD with memory in C++ and Python, and show the excellent convergence properties for top-k and random-k operators. Our scheme outperforms QSGD in progress per number of bits sent. It also opens the path to using lock-free asynchronous parallelization (e.g. Hogwild! [Niu et al. 2011]) on dense problems as the sparsity of the gradient updates is enforced.

Keywords: convex optimization, stochastic gradient descent, distributed parallel optimization, sparsification, communication efficiency.
Contents

Acknowledgment 5

Introduction 7

1 Optimization and Machine Learning 9
  1.1 Background in Convex Optimization ........................................... 9
  1.2 Notations and definitions ......................................................... 10
  1.3 Optimization in Machine Learning ............................................ 11
    1.3.1 Linear models: logistic regression .................................... 11
    1.3.2 Neural networks ............................................................... 12

2 Motivation and related work 13
  2.1 Parallel SGD ................................................................. 13
  2.2 Local SGD ................................................................. 14
  2.3 Asynchronous Parallel SGD .................................................. 14
  2.4 Addressing the communication bottleneck ................................... 15
    2.4.1 Quantized SGD ............................................................. 15
    2.4.2 Sparsification: a naive approach .................................... 15
  2.5 Sparsification in practice .................................................... 16

3 Sparsified SGD with Memory 17
  3.1 Sparsification operators and properties ................................... 17
  3.2 Algorithm ................................................................. 18
  3.3 Main contribution ............................................................ 18
  3.4 Analysis ................................................................. 19
    3.4.1 Perturbed iterate analysis ............................................ 19
    3.4.2 Bounding the error ...................................................... 19
    3.4.3 Bounding the memory ..................................................... 21
    3.4.4 Optimal estimate averaging ........................................... 23
  3.5 Extension to parallel setting .................................................. 24
  3.6 Discussion ............................................................... 26

4 Experiments 27
  4.1 Implementations ............................................................. 27
    4.1.1 Python implementation (simulation) .................................. 27
    4.1.2 C++ implementation ...................................................... 28
    4.1.3 About sparsity ............................................................. 28
    4.1.4 Python wrapping ........................................................... 28
  4.2 Datasets ................................................................. 28
  4.3 Baselines ............................................................... 29
  4.4 Results ................................................................. 29
    4.4.1 Convergence illustration .............................................. 29
    4.4.2 Comparison with QSGD ............................................... 31
    4.4.3 Multicore scale up ...................................................... 32
  4.5 Discussion ............................................................... 34
I would like to thank Martin and Sebastian for their support, their kindness and their patience with me. The freedom they gave me was a great opportunity to feed my curiosity and discover new fields. It has been a great pleasure to work closely with Sebastian and learn from his experience.

I want to thank all the Machine Learning and Optimization lab members with whom we formed a great team over the past semester. It was always exciting to come to work and discuss with them. The passion and expertise of each of them in the research they conduct inspired me. Their humility and will to share their knowledge showed me a model to follow.

Finally, I want to thank my family and my friends who were always supportive during my studies and showed interest in my work. They also have reminded me that there are more important things in life than exams, deadlines and vector spaces.
Introduction

In the past few years, Machine Learning and Deep Learning harvested impressive successes using optimization to solve hard problems in computer vision (Krizhevsky et al., 2012; Szegedy et al., 2017), neural machine translation (Vaswani et al., 2017), reinforcement learning (Gibney, 2016) and other domains. But models become more and more complex and training them is time-consuming, that is why high scale distributed optimization has gained traction recently. It becomes necessary to have the theory to back up asynchronous parallel algorithms used successfully in practice on multiple machines and GPUs (e.g. Goyal et al. (2017)). While many contributions investigated ways to distribute SGD (e.g. Niu et al. (2011); Zhang et al. (2016)), it is only more recently that researchers and practitioners focused on the communication bottleneck between workers (e.g. Lin et al. (2018); Alistarh et al. (2018)).

In this thesis, we tackle the parallel SGD problem in the convex synchronous setting with the objective to decrease the data communication without hurting the convergence. We present a simple sparsification schema that converges as fast as SGD but communicates far less (up to $\times 1000$ times less) without sacrificing the rate of convergence. Instead of dropping the coordinates that we do not share at the current iteration, we aggregate them in a memory and each of them will be eventually applied, with a delay. Our main contribution in Theorem 2 and remarks 10 states that sharing only $k = O(1)$ coordinates at each step of SGD still gives a $O(1/T)$ asymptotic rate for strongly convex and smooth functions. This indicates that scaling out the number of machines, even with limited bandwidth communication channels, is a promising strategy to solve harder and bigger Machine Learning problems.

Our method has two distinct advantages over parallel SGD. On the one hand, it decreases the size of the communicated gradients at each step, using a sparse representation of a few non-zero coordinates instead of communicating the full gradient. This allows workers to share their progress more often, bypassing the need to increase the batch size to accumulate better gradients before broadcasting. On the other hand, the enforced sparsity of the updates on the global parameters seems to open a path to applying Hogwild! (Niu et al., 2011) for asynchronous parallel sparsified SGD with memory. This second point is more speculative as it is not encompassed by our theory but our results with linear models in Python show a better scale up of our method compared to dense full gradient updates (even though it might be implementation dependent). However, sparsification has already been shown to help in practice for training bigger neural network models where the communication speed up is more apparent (e.g. Dryden et al. (2016); Aji and Heafield (2017); Lin et al. (2018); Strom (2015); Seide et al. (2014)).

This Master thesis is organized as follow: chapter 1 gives some background in Convex Optimization and Machine Learning; reader should refer to section 1.2 and Appendix A for definition and tricks used later in the proofs. We present the lines of research relative to our work in chapter 2 and demonstrate some motivation for a new communication efficient variant of SGD for distributed systems. The sparsified SGD with memory is presented in chapter 3 along with our main theorem for convergence rate and its proof. Finally, we present our implementations in Python and C++ in chapter 4 and conduct experiments to demonstrate the convergence properties of Mem-SGD and its efficient use of communication resources. Readers familiar with the field should directly consult chapters 3 and 4.
Chapter 1

Optimization and Machine Learning

Many real-world problems can be solved by minimizing a cost or a risk, or maximizing a reward. Convex optimization is a powerful tool developed to solve such problems. This chapter gives an introduction to convex optimization and how it is used in Machine Learning. Section 1.2 should be seen as a reference for the assumptions used later in chapter 3. Section 1.3.1 introduces logistic regression, a well known classification algorithm that will be used in the experimental part (chapter 4).

1.1 Background in Convex Optimization

We consider the finite sum minimization problem: given \( n \) functions \( f_i : \mathbb{R}^d \to \mathbb{R} \) for \( i \in [n] \), we want to find \( x \) that minimizes the average function value. More formally, we want to find \( x^\star \) such that,

\[
x^\star = \arg \min_x F(x), \quad F(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x).
\] (1.1)

In the convex setting, if the derivative of the function \( F \) over all its domain \( \mathbb{R}^d \) is known and can be inversed, the minimizer \( x^\star \) is defined by the constraint on its first order derivative

\[
\nabla F(x) = 0.
\] (1.2)

But in some Machine Learning applications (e.g. logistic regression, SVM, ...), there exist no closed form for \( x^\star \) from the first order constraint and we are interested in alternative algorithm to find the minimizer. A common algorithm to minimize a function is the steepest descent method, at each step we take a small step in the direction opposite to the gradient computed at the current point.

\[
x_{t+1} = x_t - \eta_t \nabla F(x_t)
\] (1.2)

where \( \eta_t \) is the step size. Given that the gradient is only a local indication of the direction with steepest slope, this step size should not be too big, otherwise the next iterate could be higher than the current and we could never reach the neighborhood of \( x^\star \). This algorithm is fully described in algorithm 1. If the constant step size is not too big and we assume strong convexity (definition 2) and smoothness (definition 3), we are guaranteed to converge after \( T = \mathcal{O} \left( \log (1/\epsilon) \right) \) to an iterate \( x_T \) such that \( F(x_T) - F(x^\star) \leq \epsilon \).

<table>
<thead>
<tr>
<th>Algorithm 1</th>
<th>Gradient-Descent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: require</td>
<td>step size ( \eta_t )</td>
</tr>
<tr>
<td>2: Initialize variable ( x_0 ) randomly</td>
<td></td>
</tr>
<tr>
<td>3: for ( t ) in 0...( T-1 ) do</td>
<td></td>
</tr>
<tr>
<td>4: ( x_{t+1} \leftarrow x_t - \eta_t \nabla F(x_t) )</td>
<td></td>
</tr>
<tr>
<td>5: end for</td>
<td></td>
</tr>
<tr>
<td>6: return ( x_T )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 2</th>
<th>Stochastic-Gradient-Descent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: require</td>
<td>step size ( \eta_t )</td>
</tr>
<tr>
<td>2: Initialize variable ( x_0 ) randomly</td>
<td></td>
</tr>
<tr>
<td>3: for ( t ) in 0...( T-1 ) do</td>
<td></td>
</tr>
<tr>
<td>4: ( i_t \leftarrow ) sample uniformly at random from ( [n] )</td>
<td></td>
</tr>
<tr>
<td>5: ( x_{t+1} \leftarrow x_t - \eta_t \nabla f_{i_t}(x_t) )</td>
<td></td>
</tr>
<tr>
<td>6: end for</td>
<td></td>
</tr>
<tr>
<td>7: return ( x_T )</td>
<td></td>
</tr>
</tbody>
</table>

The steepest descent method does not take into account that our function \( F \) is the average of \( n \) sub-functions \( f_i \)'s. This is of crucial importance in Machine Learning where \( n \) represents the number
of data points of the dataset and can be very large (typically $> 10^6$). Computing the average gradient over all these functions is expensive. Instead, we can use the Stochastic Gradient Descent algorithm (algorithm 2), which became the standard practice in Machine Learning and Deep Learning. At each step $t$, we choose uniformly at random a loss function $f_i$, and apply a step of gradient descent as follow,

$$x_{t+1} = x_t - \eta_t \nabla f_i(x_t).$$

(1.3)

Each iteration involves only one loss function (instead of $n$) and the computation is $n$ times faster. We traded some computation time at each iteration for a new source of noise from the stochasticity of the sampled loss function $f_i$. But even though the negative gradient of the sub function $f_i$ can point us in a “wrong” direction, it is an unbiased estimator of the true gradient $\nabla F(x) = E_i \nabla f_i(x)$. In other words, on average it will still lead us toward the steepest descent direction. To balance the noise in the gradient estimate, we use a decreasing step size $\eta_t$. We make a lot of progress when we are far from $x_*$ (taking big steps is not too dangerous), then we use smaller step sizes when we reach the neighborhood of $x_*$. It is a very efficient algorithm with a rate of convergence of $O(1/T)$ for smooth, strongly convex functions. In practice, it is also used to optimize non convex functions as we will see in section 1.3.2.

### 1.2 Notations and definitions

We introduce some notations and definitions that are standard in convex optimization.

**Notation 1.** We denote vectors with bold letters (e.g. $\mathbf{x}$). $x_i$ is a scalar representing the $i$th coordinate of the vector $\mathbf{x}$.

**Notation 2.** Given a function $f : \mathbb{R}^d \to \mathbb{R}$, we denote by $\mathbf{x}_*$ its minimizer (if existing and unique) and $f^*$ the value of the global minimum,

$$\mathbf{x}_* = \arg \min_{\mathbf{x}} f(\mathbf{x}), \quad f^* = f(\mathbf{x}_*).$$

(1.4)

Some proofs of convergence require assumptions on the shape and the rate of change in the neighborhood of a point of the function $f$ to optimize.

**Definition 1** (convexity). We say that a function $f : \mathbb{R}^d \to \mathbb{R}$ is convex iff

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle, \forall x, y \in \mathbb{R}^d.$$  

(1.5)

**Definition 2** ($\mu$-strong convexity). We say that a function $f : \mathbb{R}^d \to \mathbb{R}$ is $\mu$-strongly convex if

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2} \|x - y\|^2, \forall x, y \in \mathbb{R}^d.$$  

(1.6)

A corollary of the strong convexity property is that any local minimum is global because for any $x \neq \mathbf{x}_*$, we have $\|x - \mathbf{x}_*\|^2 > 0$ and $\nabla f(\mathbf{x}_*) = 0$, then $f(x) > f(\mathbf{x}_*)$.

**Definition 3** ($L$-smoothness). We say that a function $f : \mathbb{R}^d \to \mathbb{R}$ is $L$-smooth if its gradient is $L$-Lipschitz continuous,

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L \|x - y\|_2, \forall x, y \in \mathbb{R}^d$$

(1.7)

or equivalently,

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2} \|x - y\|^2, \forall x, y \in \mathbb{R}^d$$

(1.8)

An illustration of the strong convexity and the smoothness properties can be found in fig. 1.1. Observe that a strongly convex and smooth function is locally “sandwiched” between two quadratic functions. The strong convexity ensures that the function is curving enough so we make some progress toward the optimal point. The smoothness condition ensures that the function does not “jump up” next to our current point and that our next iterate becomes worst. From the previous definitions, we derive useful inequalities presented in Appendix A. Remark 11 to 13.
1.3 Optimization in Machine Learning

Many Machine Learning algorithms rely on convex optimization to learn patterns from the data. These algorithms define a parametrized model, which tries to explain some observed data, and an associated loss function, which expresses how much the observed data points deviate from this model. The goal is to minimize this loss function. We illustrate this process of modeling and derive the loss on a simple example: logistic regression. We show that the classification problem can be expressed as the minimization of a convex loss function, hence enabling the use of Stochastic Gradient Descent to optimize it. We will use this model in chapter 4 for the experimental part of this thesis.

1.3.1 Linear models: logistic regression

Consider a problem of binary classification where we want to predict the label (i.e. +1 or -1) of a given vector of input features. We are given a data matrix $A \in \mathbb{R}^{n \times d}$ where each row $a_i$ is a vector of features, and a label vector $b \in \{-1, +1\}^n$ containing the associated labels $b_i$. During the training, logistic regression learns a vector of weights $x \in \mathbb{R}^d$ that represents a linear predictor for the label of the input vector. The probability distribution of the label $b_i$ for an input vector $a_i$ according to our model is,

$$
\Pr(b_i = +1 | a_i, x) = \sigma(x^\top a_i), \\
\Pr(b_i = -1 | a_i, x) = 1 - \sigma(x^\top a_i) = \sigma(-x^\top a_i),
$$

with $\sigma$ the sigmoid function defined as $\sigma(x) = (1 + \exp(-x))^{-1}$. The sigmoid is used to map a scalar into a probability measure, $\mathbb{R} \rightarrow [0, 1]$. Furthermore, if we assume that all data points are independently sampled from the distribution defined in (1.9), the likelihood that our parametrized model generates the labels of the training dataset is the following,

$$
\mathcal{L}(A, b) = \Pr(b | A, x) = \prod_{i=1}^{n} \Pr(b_i | a_i, x) = \prod_{i=1}^{n} \sigma(b_i x^\top a_i) 
$$

We consider the negative log likelihood as loss function,

$$
\mathcal{L}(x) = \sum_{i=1}^{n} \log(1 + \exp(-b_i x^\top a_i))
$$

We want to maximize the likelihood of the observed data given our model $x$. It is equivalent to minimizing the negative log likelihood in (1.11). This loss function is a sum of convex functions in $x$ (composition of convex and linear functions) thus we can use the Stochastic Gradient Descent (SGD) algorithm to optimize it. The function $g(x) = \log(1 + \exp(-x))$ is plotted on fig. [1.3], we observe that the loss is higher when $b_i$ and $x^\top a_i$ are of opposite signs, i.e. when we misclassify the data point.

We introduce the regularization parameter $\lambda$ and the associated regularizer term in the loss function,

$$
\mathcal{L}(x) = \sum_{i=1}^{n} \log(1 + \exp(-b_i x^\top a_i)) + \frac{\lambda}{2} \|x\|_2^2
$$

This extra parameter is meant to make the problem better defined (enforced strong convexity) and encourage the model vector’s norm $\|x\|^2$ to be small. It is a tunable hyper parameter that should be chosen by cross validation on the train and evaluation dataset. In our experiment we will set $\lambda \equiv 1/n$ as advised in [Bottou] 2010.
1.3.2 Neural networks

In the past few years, Neural Networks settled down as successful building blocks for Machine Learning applications. The emergence of new differentiable programming frameworks such as Theano (Bergstra et al., 2011), Tensorflow (Abadi et al., 2016) or PyTorch (Paszke et al., 2017), enabled many researchers to easily build more complex models, train them efficiently and solve more complex problems. The main selling point of these frameworks is the automatic computation of the gradients with regard to any variables or inputs. It is also highly optimized and benefits from hardware accelerated devices such as GPUs.

The deep learning community harvested many successes for solving problems thought to resist for many more years. It surpassed the state of the art in vision (Krizhevsky et al., 2012), Neural Machine Translation (Vaswani et al., 2017), reinforcement learning (Gibney, 2016; Mnih et al., 2013), generative models (Goodfellow et al., 2014; Gulrajani et al., 2017). The models used to achieve human performance are highly non-convex, yet the Stochastic Gradient Descent is heavily used in practice.

We will not go into further depth of Neural Networks in this thesis, given that our proof focuses on the convex case and we will stick to linear models for the experiments in chapter 4. We only mention that with the apparition of bigger models, the size of the gradients to share in a multi-workers setting rocketed. It is one of the motivations for sparsiﬁed SGD with memory, to provide a sparsiﬁcation scheme that decreases the size of these exchanged gradients without hurting convergence. We hope that our current proof will help better understand what happens when we apply gradient sparsiﬁcation during neural network training.
Chapter 2

Motivation and related work

As we have seen in section 1.3, the Stochastic Gradient Descent algorithm is the workhorse for many learning tasks. However it is a highly sequential algorithm because each iterate depends on the previous one. Given the increase in problem size and the availability of bigger computing resources (multi-core processors, GPUs, clusters), the line of work of parallel and distributed optimization has shown some extensive development over the past years.

In this chapter, we introduce parallel SGD (section 2.1), in which multiple workers join their computation power at each step to compute a mini-batch gradient. In section 2.2 we present local SGD, another way to synchronization parallel workers, and asynchronous parallel SGD (i.e. Hogwild!) in section 2.3. Finally in section 2.4, we study strategies aiming at reducing the communication bottleneck arising in the distributed setting – either between machines on the networks or between GPUs on a single machine. We will present the two main strategies that appeared to alleviate this bottleneck: gradient quantization and sparsification.

2.1 Parallel SGD

With multi-core architecture, multiple steps of SGD can be run in parallel; the question is how to make the workers cooperate efficiently. One way is to run \( W \) independent instances of SGD on the same problem, maybe with different initializations. When every workers have computed \( T \) steps, the algorithm outputs the mean estimate \( \bar{x}_T = \frac{1}{W} \sum_w x_T^w \) (one shot averaging) or the best one, \( \hat{x}_T = \arg \min \{x_T^w : w \in [W]\} f(x_T^w) \).

Another approach is to synchronize all the workers at each step, let them share their gradients and agree on the next iterate as in the following update equations,

\[
x_{t+1} := x_t - \eta t \sum_{w=1}^{W} g_t^w, \quad g_t^w := \eta t \nabla f_i (x_t) \quad (2.1)
\]

This setting is very close to mini-batch SGD, where the single worker computes the gradient on a batch of functions \( f_i \)'s instead of only one,

\[
x_{t+1} := x_t - \eta t \left( \frac{1}{|I_t|} \sum_{i \in I_t} \nabla f_i (x_t) \right), \quad (2.2)
\]

where \( I_t \subseteq [n] \) is a subset of indexes of the \( n \) loss functions \( f_i \)'s picked uniformly at random. The size of the batch in parallel SGD is the number of workers (i.e. \( |I_t| \equiv W \)) and the variance of the stochastic gradient estimator is reduced by a factor \( W \), hence, it gives a linear decrease of number iteration when the number of workers increases. Unfortunately, at each round we are forced to wait for the slowest worker to share its gradient, which in practice hurts dramatically the speed of convergence (in term of time, not iteration) as workers are idle most of the time.

We would like to highlight the two main (distinct) issues that arise in parallel SGD:

- **synchronization lock**: all the gradients are needed to compute the averaged gradient, then the end of each step depends on the slowest worker,
gradient communication: if the workers do not use shared memory, they need to transmit the full gradient of the parameters that can be of a consequent size and slow down the process. These two orthogonal problems are addressed with different approaches in the following algorithms. Local SGD and Asynchronous SGD try to improve the cooperation between the workers, while QSGD tackles the communication bottleneck.

2.2 Local SGD

To avoid to synchronize too often, local SGD \cite{Stich2018} proposes to let each worker follow \( L \) local steps of SGD between each synchronization. The update steps are defined as follow,

\[
x^{w}_{t,l+1} := x^{w}_{t,l} - \eta^{t}_{l} \sum_{i \in I^{w}_{t,l}} \nabla f_{i}(x^{w}_{t,l}), \quad 0 < l < L,
\]

\[
x^{w}_{t,0} := x^{w}_{t,0} + \frac{1}{W} \sum_{v=1}^{W} (x^{v}_{t,L} - x^{v}_{t,0}).
\]

with \( L \) the number of local updates between workers’ synchronization and \( B := |I^{w}_{t,l}| \) the fixed batch size of local updates. Equation (2.3) denotes the local updates that are performed independently on each worker \( w \), while eq. (2.4) is the synchronization step (blocking) between workers where we average the progress of all the workers over their last \( L \) local steps. Observe that the hyperparameter \( L \) provides a way to balance between one shot SGD (\( L = T \)) and parallel SGD (\( L = 1 \)).

It has been shown that this schema works well in practice, even on non-convex problems. The recent theoretical analysis proposed by \cite{Stich2018} shows asymptotic rate of convergence comparable to parallel SGD, i.e. \( O(\sigma^{2}/\mu WT) \) with \( \sigma^{2} := E_{i} \| \nabla f_{i}(x) - \nabla f(x) \|^{2} \). This is interesting because this cooperation scheme requires less inter worker synchronization but still exhibits a \( W \) speed up in the main term of the rate.

2.3 Asynchronous Parallel SGD

To overcome the synchronization limitations of parallel SGD, some implementations do not wait for each worker to compute its gradient at each step to compute the average (mini-batch), but rather let each worker directly read and update a shared version of the parameters to optimize (e.g. \( x \)). This raises two concerns:

- **stale gradients**: the parameter \( x \) can be updated by other workers during the computation of the gradient by a worker. The computed gradient is then applied on another iterate than the one it was computed on. For a delay of \( \tau \) steps where other workers updated the shared parameter, the update becomes

\[
x_{t+\tau+1} = x_{t+\tau} - \eta \nabla f_{i}^{w}(x_{t})
\]

- **inconsistent reads**: given that each worker does not update all the coordinates of \( x \) atomically (otherwise it would have to acquire a lock and the benefits of asynchronous parallelism would be lost), it can happen that a worker reads the shared parameter \( x \) when another worker has only updated a subset of the coordinates, the reader views a mix of \( x_{t} \) and \( x_{t+1} \).

The first theoretical analysis of this scheme, that was known to work in practice, is due to \cite{Niu2011} with the Hogwild! algorithm. It builds upon the fact that the work can be parallelized if the problem is sparse, which is the case for common problems such as SVM, matrix factorization or graph cut, because conflicting access to the same coordinates of the parameters will be rare. However, it does not provide bounds for dense problems, such as logistic regression on a dense dataset or non-convex cases, such as deep neural networks.

More recently this idea of asynchronous parameter update has been applied to deep learning. The parameter server became a standard architecture: workers query the server to get the last set of
parameters and notify it when they want to apply an update. It solves the inconsistent read problem but the stale gradient issue remains. Chen et al. (2016) show good performance in practice on ImageNet yielding almost linear speed up when increasing the number of workers from 25 to 200. Another interesting line of work is to correct the error introduced by the delay before applying the gradient, for example Zheng et al. (2017) uses some second-order information to fix the stale gradient.

Using lock-free updates can solve the problem of synchronization between workers, where most workers stay idle waiting for the last one to finish its computation. The new bottleneck now occurs in the communication of the updates between the cores, the GPUs or even the different machines in a distributed set up.

2.4 Addressing the communication bottleneck

All the previously introduced schema to parallelize SGD rely on the communication of the updates between the workers. Even though we can reduce these synchronization barriers, the time spent sending the gradients between workers over the networks or between GPUs is non negligible. We then encounter a new bottleneck: the communication time.

2.4.1 Quantized SGD

A promising line of work to address the communication bottleneck is to compress the gradients before aggregating them. For instance, quantizing each single precision scalar into a INT8 saves a factor 4 of data quantity to transfer. Seide et al. (2014) proposed an even more radical approach: sending only the sign of each coordinate and the norm of the gradient (32 + d bits). They empirically show that this ultra compression method still converges for deep learning problems.

A maybe more sensible approach taken by Alistarh et al. (2017) is to define a number of level of quantization $s$ and share the norm of the gradient $\|g\|$, the sign of each coordinate $g_i$ and the quantization level $l_i$ of $|g_i|$. The crucial property of the quantization they propose is the unbiased property,

$$\mathbb{E} Q_s(g) = g. \quad (2.6)$$

Implementing this quantization scheme in practice is easy, we randomly assign a scalar to the previous or next level proportionally to the barycenter coefficient of the point (i.e. the closer, the more likely). It distinguishes from the deterministic casting that can be done with processor instructions because this operator usually takes the closest lower bound of each coordinate. To share a vector $g$, a worker will send the triple $(\|g\|, \text{sgn}(g), l)$.

An extra step can be taken in order to compress the levels $l_i$’s. Given that small levels are more likely to appear, Elias coding (Elias, 1975) is used to encode these integers and reduce the number of bits used. Another improvement comes from the sparsity of the level representation: for $s = \sqrt{d}$ approximately $\sqrt{d}$ coordinates will be mapped to $l_i = 0$ which let us share a sparse gradient (non-zero indexes and values) instead of the dense one. All compression strategies applied together and using the advised $s = \sqrt{d}$ levels. Alistarh et al. (2017) Corollary 3.3) shows existence of an encoding scheme with expected length at most $2.8d + 32$ bits. It is an order of magnitude smaller than the single precision dense gradient ($32d$ bits).

2.4.2 Sparsification: a naïve approach

Another tempting approach to reduce the size of the communicated gradient is to share only some of its coordinates. This is what we study in the next chapter and for motivation, we first follow a naïve approach where the non-shared coordinates are simply dropped. Consider the following variant of SGD, where $d - k$ random coordinates of the stochastic gradient are dropped:

$$x_{t+1} = x_t - \eta_t g_t, \quad g_t := \frac{d}{k} \cdot \text{rand}_k(\nabla f_i(x_t)), \quad (2.7)$$

where $i \sim \text{a.a.r.}[n]$ and rand$_k$ selects only $k$ coordinates. It is important to note that the update is unbiased, i.e. $\mathbb{E} g_t = \nabla f(x)$. For carefully chosen stepsizes $\eta_t$ this algorithm converges at rate $O\left(\frac{s^2}{T}\right)$
on strongly convex and smooth functions $f$, where $\sigma^2$ is an upper bound on the variance, see for instance [Zhao and Zhang (2015)]. We have

$$\sigma^2 = \mathbb{E} \| \frac{d}{k} \text{rand}_k (\nabla f_i (x)) - \nabla f (x) \|^2 \leq \mathbb{E} \| \frac{d}{k} \text{rand}_k (\nabla f_i (x)) \|^2 \leq \frac{d}{k} \mathbb{E} \| \nabla f_i (x) \|^2 \leq \frac{d}{k} G^2$$

where we used the variance decomposition $\mathbb{E} \| X - \mathbb{E} X \|^2 = \mathbb{E} \| X \|^2 - \| \mathbb{E} X \|^2$ and the standard assumption $\mathbb{E} \| \nabla f_i (x) \|^2 \leq G^2$. Hence, when $k$ is small this algorithm requires $d$ times more iterations to achieve the same error guarantee as vanilla SGD.

It is well known that by using mini-batches the variance of the gradient estimator can be reduced. If we consider in (2.7) the estimator $g_t := \frac{d}{k} \cdot \text{rand}_k \left( \frac{1}{\tau} \sum_{i \in I_{\tau}} \nabla f_i (x_t) \right)$ for $\tau = \lceil \frac{k}{d} \rceil$, and $I_{\tau} \sim \text{u.a.r.} \left( \binom{n}{k} \right)$ instead, we have

$$\sigma^2 = \mathbb{E} \| g_t - \nabla f (x_t) \|^2 \leq \mathbb{E} \| \frac{d}{k} \cdot \text{rand}_k \left( \frac{1}{\tau} \sum_{i \in I_{\tau}} \nabla f_i (x_t) \right) \|^2 \leq \frac{d}{k \tau} \mathbb{E} \| \nabla f_i (x_t) \|^2 \leq G^2.$$  

(2.8)

This shows that, when using mini-batches of appropriate size, the sparsification of the gradient does not hurt convergence. However, by increasing the mini-batch size, we increase the computation by a factor of $\frac{d}{k}$.

These two observations seem to indicate that the factor $\frac{d}{k}$ is inevitably lost, either by the increased number of iterations or the increased computation. However, this is no longer true when the information in (2.7) is not dropped but kept in memory. We will indeed demonstrate below that sparsified SGD with memory converges at the same rate as vanilla SGD.

### 2.5 Sparsification in practice

The first form of gradient sparsification was proposed by [Strom (2015)] using a residual accumulation to eventually use all computed gradients. To select which coordinates should be shared, they used a fixed threshold $\tau$ which had to be tuned. They further quantized the coordinates by sharing only the sign of meaningful coordinates, sending 0 if $|x_i| > \tau$, 1 if $|x_i| < \tau$ and not sending coordinates with smaller magnitudes (known as 1-bit SGD).

[Dryden et al. (2016)] proposed to tune the threshold $\tau$ to share a fixed proportion of coordinates at each iteration. [Aji and Henfield (2017)] also used a similar scheme to train on MNIST and NMT tasks. The sparsification methods used in Deep Learning were further improved by [Lin et al. (2018)] with Deep Gradient Compression. They describe a bag of tricks to make sparsification work better in practice and obtain impressive results on ImageNet cutting the gradient size of ResNet-50 from 97MB to 0.35MB without loss of accuracy. They used momentum correction, gradient clipping and warm-up training to overcome some of the issues introduced by the sparsification.

Gradient sparsification works well in practice, especially for complex models that need to share a large amount of data to propagate the updates. However, it seems that many tricks need to be applied to overcome the delay of application of some gradients. Our theory, even if restricted to convex function, tries to explain the consequences of these delay and how learning rate should be tuned to mitigate this problem.
Chapter 3

Sparsified SGD with Memory

In this chapter, we present sparsified SGD with memory, a variant of SGD which only updates a small subset of coordinates at each step but still converges at the same rate as vanilla SGD. The advantage of these sparse updates is twofold: it drastically reduces the amount of data needed to be shared between workers in the parallel set up (see section 3.5) and it decreases the conflicts when multiple workers cooperate in shared memory (see chapter 4).

In section 3.1, we define the sparsification operators that are used in the algorithm. We will describe the algorithm in section 3.2 and prove that the asymptotic convergence rate in the sequential setting is the same as SGD (section 3.4). An extension to the parallel setting and some discussions about the step size are presented in sections 3.5 and 3.6.

3.1 Sparsification operators and properties

In sparsified SGD, only a subset of the coordinates of the gradient are shared ($k$ among $d$). We will consider the following two $k$-sparsification operators, rand-$k$ and top-$k$, defined as follows:

**Definition 4 (top-$k$).** For a parameter $1 \leq k \leq d$, the operators $\text{top}_k : \mathbb{R}^d \rightarrow \mathbb{R}^d$ is defined for $x \in \mathbb{R}^d$ as

$$\left(\text{top}_k(x)\right)_i := \begin{cases} (x)_{\pi(i)}, & \text{if } i \leq k, \\ 0, & \text{otherwise,} \end{cases}$$

(3.1)

where $\pi$ is a permutation of $[d]$ such that $(|x|)_{\pi(i)} \geq (|x|)_{\pi(i+1)}$ for $i = 1, \ldots, d-1$.

**Definition 5 (rand-$k$).** For a parameter $1 \leq k \leq d$, the operators $\text{rand}_k : \mathbb{R}^d \times \Omega_k \rightarrow \mathbb{R}^d$, where $\Omega_k = \left(\binom{d}{k}\right)$ denotes the set of all $k$ element subsets of $[d]$, is defined for $x \in \mathbb{R}^d$ as

$$\left(\text{rand}_k(x, \omega)\right)_i := \begin{cases} (x)_i, & \text{if } i \in \omega, \\ 0, & \text{otherwise.} \end{cases}$$

(3.2)

We abbreviate $\text{rand}_k(x)$ whenever the second argument is chosen uniformly at random, $\omega \sim_{\text{u.a.r.}} \Omega_k$.

We also define the $k$-contraction property that will be necessary in the proof of the rate of convergence. This property ensures that removing $k$ coordinates of a vector $x$ according to a sparse$_k$ operator, reduces the norm of this vector by at least $(1 - k/d)$ in average.

**Definition 6 ($k$-contraction property).** For a parameter $0 < k \leq d$, a (random) operator $\text{sparse}_k : \mathbb{R}^d \rightarrow \mathbb{R}^d$ that satisfies the contraction property

$$\mathbb{E} \|x - \text{sparse}_k(x)\|^2 \leq \left(1 - \frac{k}{d}\right) \|x\|^2,$$

(3.3)

for all $x \in \mathbb{R}^d$ is a $k$-contraction operator.
We now state the main convergence theorem for sparsified SGD with memory as defined above.

Algorithm 3

Let $\omega_k$.

Theorem 2.

Note that the gradients get multiplied with the step size $\eta$, which concludes the proof.

3.3 Main contribution

We define the following update rules for parameter $0 < k \leq d$ and $k$-sparsification operator $\text{sparse}_k: \mathbb{R}^d \rightarrow \mathbb{R}^d$ satisfying the $k$-contraction property (cf. definition [6]):

\[
x_{t+1} := x_t - g_t
\]

\[
g_t := \text{sparse}_k (m_t + \eta_t \nabla f_i(x_t))
\]

\[
m_{t+1} := m_t + \eta_t \nabla f_i(x_t) - g_t
\]

where $i_t \sim_{\text{unif}} [n]$, $m_0 = 0$ and $\{\eta_t\}_{t \geq 0}$ denotes a sequence of stepsizes. The pseudocode is given in Algorithm 3. Note that the gradients get multiplied with the step size $\eta_t$ at the time step $t$ when they are put into memory, and not when they are (partially) retrieved from it. You can find an example of the sequence of iterates $x_t$ generated by a run of Mem-SGD in fig. 3.1.

Algorithm 3 Mem-SGD

1. Initialize variables $x_0$ and $m_0 = 0$
2. for $t$ in $0 \ldots T - 1$
3. Sample $i_t$ uniformly in $[n]$
4. $g_t := \text{sparse}_k (m_t + \eta_t \nabla f_i(x_t))$
5. $x_{t+1} := x_t - g_t$
6. $m_{t+1} := m_t + \eta_t \nabla f_i(x_t) - g_t$
7. end for

3.3 Main contribution

We now state the main convergence theorem for sparsified SGD with memory as defined above.

Theorem 2. Let $f_i$ be $L$-smooth, $f$ be $\mu$-strongly convex, $0 < k \leq d$, $\mathbb{E}_t \| \nabla f_i(x_t) \|_2^2 \leq G^2$ for $t = 0, \ldots, T - 1$, where $\{x_t\}_{t \geq 0}$ are generated according to (3.7) for stepsizes $\eta_t = \frac{2}{\mu (a + t^2)}$ and shift parameter $a > 1$. Then for $\alpha > 4$ such that $\frac{(\alpha + 1) \rho + \rho^2}{\mu + 1} \leq a$, with $\rho := \frac{\alpha}{\alpha - 4}$, it holds

\[
\mathbb{E} f(x_T) - f^* \leq \frac{4T(T + 2a \sqrt{\mu} \rho + \mu a^2 \rho^2)}{\mu S_T} G^2 + \frac{\mu a^3 \rho^3}{8 S_T} \| x_0 - x^* \|^2 + \frac{64 T (1 + 2 \frac{\rho}{\mu})}{\mu S_T} \left( \frac{4 \alpha}{\alpha - 4} \right) \frac{d^2}{k^2} G^2,
\]

where $\bar{x}_T = \frac{1}{S_T} \sum_{t=0}^{T-1} w_t x_t$, for $w_t = (a + t)^2$, and $S_T = \sum_{t=0}^{T-1} w_t \geq \frac{1}{2} T^3$. 

18
3.4 Analysis

We give the proof of theorem 2, it is a reorganized version of the proof in our submitted article currently in review for NIPS 2018.

3.4.1 Perturbed iterate analysis

Inspired by the perturbed iterate framework in Mania et al. (2017) and Leblond et al. (2017) we first define a virtual sequence \( \{ \tilde{x}_t \}_{t \geq 0} \) in the following way:

\[
\tilde{x}_0 = x_0, \quad \tilde{x}_{t+1} = \tilde{x}_t - \eta_t \nabla f_i(x_t),
\]

(3.9)

where the sequences \( \{x_t\}_{t \geq 0}, \{\eta_t\}_{t \geq 0} \) and \( \{i_t\}_{t \geq 0} \) are the same as in (3.7). Notice that the difference between the virtual sequence \( \tilde{x}_t \) and \( x_t \) is always equal to the part of the gradients that have not been applied yet, that is what is stored in the memory. This is visible in fig. 3.2 and can be formally derived as follow,

\[
x_t - \tilde{x}_t = \left( x_0 - \sum_{j=0}^{t-1} g_j \right) - \left( x_0 - \sum_{j=0}^{t-1} \eta_j \nabla f_i(x_j) \right) = m_t.
\]

(3.10)

Note that this sequence \( \{\tilde{x}_t\}_{t \geq 0} \) is only used for our analysis and does not appear in the implementation.

3.4.2 Bounding the error

Our goal is to upper bound the function error \( e_t := \mathbb{E} f(x_t) - f^* \). We use the perturbed iterate framework and follow a similar approach as in Leblond et al. (2017) to obtain lemma 3. The bound on \( e_t \) is more explicit in corollary 4.

**Lemma 3.** Let \( \{x_t\}_{t \geq 0} \) and \( \{\tilde{x}_t\}_{t \geq 0} \) be defined as in (3.7) and (3.9) and let \( f_i \) be \( L \)-smooth and \( f \) be \( \mu \)-strongly convex with \( \mathbb{E}_t \| \nabla f_i(x_t) \|^2 \leq G^2 \). Then

\[
\mathbb{E} \| \tilde{x}_{t+1} - x^* \|^2 \leq \left( 1 - \frac{\eta_t \mu}{2} \right) \mathbb{E} \| \tilde{x}_t - x^* \|^2 + \eta_t^2 G^2 - \eta_t e_t + \eta_t (\mu + 2L) \mathbb{E} \| m_t \|^2,
\]

(3.11)

where \( e_t := \mathbb{E} f(x_t) - f^* \).
Proof. Using the update equation (3.9) we have

\[ \| \tilde{x}_{t+1} - x^* \|^2 = \| \tilde{x}_t - x^* \|^2 + \eta_t^2 \| \nabla f_i(x_t) \|^2 - 2\eta_t \langle x_t - x^*, \nabla f_i(x_t) \rangle + 2\eta_t \langle x_t - \tilde{x}_t, \nabla f_i(x_t) \rangle . \]  

(3.12)

And by applying expectation

\[ E_{\xi_t} \| \tilde{x}_{t+1} - x^* \|^2 \leq \| \tilde{x}_t - x^* \|^2 + \eta_t^2 G^2 - 2\eta_t \langle x_t - x^*, \nabla f(x_t) \rangle + 2\eta_t \langle x_t - \tilde{x}_t, \nabla f(x_t) \rangle . \]  

(3.13)

To upper bound the third term, we use the same estimates as in (Leblond et al., 2017, Appendix C.3):

\[ -\langle x_t - x^*, \nabla f(x_t) \rangle \leq -\langle f(x_t) - f^* \rangle - \frac{\mu}{2} \| x_t - x^* \|^2 \]  

(3.14)

and with \( \| a + b \|^2 \leq 2 \| a \|^2 + 2 \| b \|^2 \) we further have

\[ -\| x_t - x^* \|^2 \leq \| x_t - \tilde{x}_t \|^2 - \frac{1}{2} \| \tilde{x}_t - x^* \|^2 . \]  

(3.15)

Putting these two estimates together, we can bound (3.13) as follows:

\[ E_{\xi_t} \| \tilde{x}_{t+1} - x^* \|^2 \leq \left( 1 - \frac{\eta_t \mu}{2} \right) \| \tilde{x}_t - x^* \|^2 + \eta_t^2 G^2 - 2\eta_t \langle x_t - x^*, \nabla f(x_t) \rangle + 2\eta_t \langle x_t - \tilde{x}_t, \nabla f(x_t) \rangle , \]  

(3.16)

where \( e_t = E f(x_t) - f^* \). We now estimate the last term. As each \( f_i \) is \( L \)-smooth also \( f \) is \( L \)-smooth, i.e. satisfies \( f(x) - f(y) - \langle \nabla f(y), x - y \rangle \geq \frac{1}{2L} \| \nabla f(y) - \nabla f(x) \|^2 \). Together with \( 2 \langle a, b \rangle \leq \gamma \| a \|^2 + \gamma^{-1} \| b \|^2 \) we have

\[ \langle x_t - \tilde{x}_t, \nabla f(x_t) \rangle \leq \frac{1}{2} \left( 2L \| x_t - \tilde{x}_t \|^2 + \frac{1}{2L} \| \nabla f(x_t) \|^2 \right) \]  

(3.17)

\[ = L \| x_t - \tilde{x}_t \|^2 + \frac{1}{4L} \| \nabla f(x_t) - \nabla f(x^*) \|^2 \]  

(3.18)

\[ \leq L \| x_t - \tilde{x}_t \|^2 + \frac{1}{2} (f(x_t) - f^*) . \]  

(3.19)
Combining with (3.10) we have
\[ \mathbb{E}_t \| \tilde{x}_{t+1} - x^* \|^2 \leq \left( 1 - \frac{\eta t \mu}{2} \right) \| \tilde{x}_t - x^* \|^2 + \eta^2 G^2 - \eta t e_t + \eta t (\mu + 2L) \| x_t - \tilde{x}_t \|^2, \]
and the claim follows with (3.10).

We rewrite the lemma as follow to make the bound on \( e_t \) appear.

**Corollary 4.** Let \( \{x_t\}_{t \geq 0} \) and \( \{\tilde{x}_t\}_{t \geq 0} \) be defined as in (3.7) and (3.9) and let \( f_i \) be \( \mu \)-strongly convex and \( f \) be \( \mu \)-smooth with \( \mathbb{E}_t \| \nabla f_i(x_t) \|^2 \leq G^2 \). Then
\[ e_t \leq \frac{1}{\eta t} \left( 1 - \frac{\eta t \mu}{2} \right) a_t - a_{t+1} + \eta^2 G^2 + \eta t (\mu + 2L) \mathbb{E} \| m_t \|^2 \]
where \( e_t := \mathbb{E} f(x_t) - f^* \) and \( a_t := \mathbb{E} \| \tilde{x}_t - x_t \|^2 \).

### 3.4.3 Bounding the Memory

From equation (3.21) it becomes clear that we should derive an upper bound on \( \mathbb{E} \| m_t \|^2 \). Before deriving the bound on the memory, observe that at each step of the Mem-SGD algorithm, the memory vector \( m_t \) ingests a new stochastic gradient \( \eta_t \nabla f_i(x_t) \) and gets some coordinates removed (and applied to \( x_t \)) by the \text{sparse}_k \) operator. After adding the new stochastic gradient we have \( v_t := m_t + \eta_t \nabla f_i(x_t) \) and after retrieving some part, \( m_{t+1} := v_t - \text{sparse}_k(v_t) \). This outlines the contraction property (3.3) of the sparsity operator \( \text{sparse}_k \) that will be used to show that \( \mathbb{E} \| m_t \|^2 = O(\eta^2 G^2) \).

We use two distinct ways to bound the memory in eq. (3.27). The first one uses the update strategy to show a shrinking relationship between \( \mathbb{E} \| m_{t+1} \|^2 \) and \( \mathbb{E} \| m_t \|^2 \). Unfortunately this recursion is applicable only if \( \eta t \equiv 8(\mu(t + a))^{-1} \) decreases slowly, i.e. for big \( t \). We use another bound for smaller \( t \). The second upper bound starts with an empty memory and only adds a stochastic gradient (whose norm is bounded by \( G^2 \) by assumption) at each step without retrieving any coordinates. It will eventually become a very loose upper bound but it is useful for small \( t \) (i.e. \( t < t_0 \)).

**Lemma 5.** Let \( \{x_t\}_{t \geq 0} \) as defined in (3.7) for \( 0 < k \leq d \), \( \mathbb{E}_t \| \nabla f_i(x_t) \|^2 \leq G^2 \) and stepsizes \( \eta_t = \frac{8}{\mu(\alpha + t)} \) with \( \alpha > 4 \), as in Theorem 2. Then
\[ \mathbb{E} \| m_t \|^2 \leq \frac{4 \alpha}{\alpha - 4} \frac{d^2}{k^2} G^2. \]

**Proof.** First, observe that by Lemma 1 and \( \| a + b \|^2 \leq (1 + \gamma) \| a \|^2 + (1 + \gamma^{-1}) \| b \|^2 \) for \( \gamma > 0 \) we have
\[ \mathbb{E} \| m_{t+1} \|^2 \leq \left( 1 - \frac{k}{4} \right) \| m_t - \eta \nabla f_i(x_t) \|^2 \]
\[ \leq \left( 1 - \frac{k}{d} \right) \left( \left( 1 + \frac{k}{2d} \right) \mathbb{E} \| m_t \|^2 + \left( 1 + \frac{2d}{k} \right) \eta^2 \mathbb{E} \| \nabla f_i(x_t) \|^2 \right) \]
\[ \leq \left( 1 - \frac{k}{2d} \right) \mathbb{E} \| m_t \|^2 + \frac{2d}{k} \eta^2 G^2. \]

On the other hand, from \( \sum_{i=1}^a a_i \|^2 \leq s \sum_{i=1}^a \| a_i \|^2 \) we also have
\[ \mathbb{E} \| m_{t+1} \|^2 \leq (t + 1) \sum_{i=0}^T \eta^2 G^2. \]

Now the lemma follows from Lemma 6 just below with \( A = \frac{8G^2}{\mu} \).
Lemma 6. Let $A \geq 0$, $d \geq k \geq 1$, $\{h_t\}_{t \geq 0}$, $h_t \geq 0$ be a sequence satisfying

\[ h_0 = 0, \quad h_{t+1} \leq \min \left\{ \left( 1 - \frac{k}{2d} \right) h_t + \frac{2d}{k} \eta_t^2 A, \ (t + 1) \sum_{i=0}^{T} \eta_i^2 A \right\}, \quad (3.27) \]

for a sequence $\eta_t = \frac{1}{a+t}$ with $a \geq \frac{(a+1)^{\frac{d}{2}} + \rho + 1}{\rho}$, for $\alpha > 4$, $\rho := \frac{4a}{a-4(a+1)}$. Then

\[ h_t \leq \frac{4a}{\alpha - 4} \eta_t^2 \frac{d^2}{k^2} A, \quad (3.28) \]

for $t \geq 0$.

Proof. The claim holds for $t = 0$.

Large $t$. Let $t_0 = \max\{\lfloor \alpha \frac{d}{2} - a \rfloor, 0\}$, i.e. $\eta_0 \leq \frac{k}{a}$. (Note that for any $a \geq \frac{k}{4}$ it holds $t_0 = 0$.) Suppose the claim holds for $t \leq t_0$. Observe,

\[ \eta_t^2 \left( 1 - \frac{2k}{ad} \right) \leq \eta_{t+1}^2, \quad (3.29) \]

for $t \geq t_0$. This follows from fact fact [4] with $c = \frac{ad}{2}$. By induction,

\[ h_{t+1} \leq \left( 1 - \frac{k}{2d} \right) \frac{4a}{\alpha - 4} \eta_t^2 \frac{d^2}{k^2} A + \frac{2d}{k} \eta_t^2 A \]

\[ = \eta_t^2 \left( 1 - \frac{2k}{ad} \right) \frac{4a}{\alpha - 4} \frac{d^2}{k^2} A, \quad (3.30) \]

where we used $t \geq t_0$ (and the observation just above) for the last inequality.

Small $t$. Assume $t_0 \geq 1$, otherwise the claim follows from the part above. We have

\[ h_t \leq \sum_{i=0}^{t-1} \eta_i^2 A \leq \frac{t}{a-1} A, \quad (3.32) \]

where we used

\[ \sum_{i=0}^{t-1} \eta_i^2 \leq \sum_{i=0}^{\infty} \frac{1}{(a+t)^2} \leq \int_{a-1}^{\infty} \frac{1}{x^2} dx = \frac{1}{a-1}, \quad (3.33) \]

for $a > 1$. For $t \leq t_0$ we have

\[ \eta_t^2 \frac{d^2}{k^2} \geq \eta_0^2 \frac{d^2}{k^2} = \frac{1}{(a+t_0)^2} \frac{d^2}{k^2} \geq \frac{1}{(\alpha+1)^2} \frac{d^2}{k^2} \geq \frac{1}{(\alpha+1)^2} \frac{d^2}{(\alpha+1)^2} = \frac{1}{(\alpha+1)^2}, \quad (3.34) \]

using $\frac{d}{k} \geq 1$. Observe $t_0 \leq \alpha \frac{d}{k} - a + 1 \leq (\alpha + 1) \frac{d}{k} - a$. For $t \leq (\alpha + 1) \frac{d}{k} - a$ we have

\[ h_t \leq \frac{t}{a-1} A \leq \frac{\alpha + 1}{\alpha - a} A - A \leq \rho A, \quad (3.35) \]

by the condition on $a$. Hence, by combining these observations,

\[ h_t \leq \frac{t}{a-1} A \leq \rho A = \frac{4a}{\alpha - 4} \frac{1}{\alpha(a+1)^2} A \leq \frac{4a}{\alpha - 4} \frac{d^2}{k^2} A \leq \frac{4a}{\alpha - 4} \eta_t^2 \frac{d^2}{k^2} A, \quad (3.36) \]

and the proof follows. \qed
3.4.4 Optimal estimate averaging

As discussed by Lacoste-Julien et al. (2012), Shamir and Zhang (2013), Rakhlin et al. (2012), we have to define a suitable averaging scheme for the iterates \( \{x_t\}_{t \geq 0} \) to get the optimal convergence rate. In contrast to Lacoste-Julien et al. (2012) that use linearly increasing weights, we use quadratically increasing weights, as for instance Shamir and Zhang (2013).

**Lemma 7.** Let \( \{a_t\}_{t \geq 0}, \alpha_t \geq 0, \{\epsilon_t\}_{t \geq 0}, \epsilon_t \geq 0, \) be sequences satisfying

\[
a_{t+1} \leq \left( 1 - \frac{\mu \eta_t}{2} \right) a_t + \eta_t^2 A + \eta_t^3 B - \eta_t \epsilon_t, \tag{3.37}
\]

for \( \eta_t = \frac{8}{\mu(a+t)} \) and constants \( A, B \geq 0, \mu > 0, a > 1. \) Then

\[
\frac{1}{S_T} \sum_{t=0}^{T-1} w_t \epsilon_t \leq \frac{\mu a^3}{8S_T} a_0 + \frac{4T(T+2a)}{\mu S_T} A + \frac{64T}{\mu^2 S_T} B, \tag{3.38}
\]

for \( w_t = (a + t)^2 \) and \( S_T := \sum_{t=0}^{T-1} w_t = \frac{T}{\mu} (2T^2 + 6aT - 3T + 6a^2 - 6a + 1) \geq \frac{1}{3} T^3. \)

**Proof.** Observe

\[
\left( 1 - \frac{\mu \eta_t}{2} \right) \frac{w_t}{\eta_t} = \left( \frac{a + t - 4}{a + t} \right) \frac{\mu(a+t)^3}{8} = \frac{\mu(a+t-4)(a+t)^2}{8} \leq \frac{\mu(a+t-1)^3}{8} = \frac{w_{t-1}}{\eta_{t-1}}, \tag{3.39}
\]

where the inequality is due to

\[
(a + t - 4)(a + t)^2 = (a + t - 1)^3 + 1 - 3a - a^2 - 3t - 2at - t^2 \leq (a + t - 1)^3, \tag{3.40}
\]

for \( a \geq 1, t \geq 0. \)

We now multiply equation (3.37) with \( \frac{w_t}{\eta_t} \), which yields

\[
a_{t+1} \frac{w_t}{\eta_t} \leq \left( 1 - \frac{\mu \eta_t}{2} \right) \frac{w_t}{\eta_t} a_t + \eta_t \epsilon_t A + \eta_t^2 B - \eta_t \epsilon_t, \tag{3.41}
\]

and by recursively substituting \( a_t \frac{w_{t-1}}{\eta_{t-1}} \) we get

\[
a_T \frac{w_{T-1}}{\eta_{T-1}} \leq \left( 1 - \frac{\mu \eta_t}{2} \right) \frac{w_0}{\eta_0} a_0 + \sum_{t=0}^{T-1} \frac{w_t \eta_t}{\eta_0} A + \sum_{t=0}^{T-1} \frac{w_t \eta_t^2}{\eta_0^2} B - \sum_{t=0}^{T-1} w_t \epsilon_t, \tag{3.42}
\]

i.e.

\[
\sum_{t=0}^{T-1} w_t \epsilon_t \leq \frac{w_0}{\eta_0} a_0 + \sum_{t=0}^{T-1} \frac{w_t \eta_t}{\eta_0} A + \sum_{t=0}^{T-1} \frac{w_t \eta_t^2}{\eta_0^2} B. \tag{3.43}
\]

We will now derive upper bounds for the terms on the right hand side. We have

\[
\frac{w_0}{\eta_0} = \frac{\mu a^3}{8}, \tag{3.44}
\]

\[
\sum_{t=0}^{T-1} \frac{w_t \eta_t}{\mu} = \sum_{t=0}^{T-1} 8(a + t) \frac{8(a + t) + 4T(T + 2a)}{\mu} \leq \frac{4T(T + 2a)}{\mu}, \tag{3.45}
\]

and

\[
\sum_{t=0}^{T-1} \frac{w_t \eta_t^2}{\mu^2} = \sum_{t=0}^{T-1} \frac{64T}{\mu^2}. \tag{3.46}
\]
Let $S_T := \sum_{t=0}^{T-1} w_t = \frac{T}{T} \left( 2T^2 + 6aT - 3T + 6a^2 - 6a + 1 \right)$. Observe
\[
S_T \geq \frac{1}{3} T^3 + aT^2 - \frac{1}{2} T^2 + a^2T - aT \geq \frac{1}{3} T^3.
\] (3.47)
for $a \geq 1, T \geq 0$.

Proof of Theorem 3.2. The proof immediately follows from lemmas 5 to 7 and convexity of $f$, i.e. we have $E f(\bar{x}_T) - f^* \leq \frac{1}{s_T} \sum_{t=0}^{T-1} w_t \epsilon_t$ in (3.38), for constants $A = G^2$ and $B = (\mu + 2L) \frac{d_0}{\eta_0^2} G^2$.

3.5 Extension to parallel setting

We further study how sparsified SGD with memory can be parallelized with $W$ workers. Recall that for parallel SGD (section 2.1), we average the $W$ computed gradients to get an estimate of the full gradient which has a decreased variance by a factor $W$. That is,
\[
E \left\| \frac{1}{W} \sum_{w=1}^{W} \nabla f_{i_w}(x_t) - \nabla f(x_t) \right\|^2 \leq \frac{\sigma^2}{W},
\] (3.48)
with $\sigma^2 := E_i \| \nabla f_i(x) - \nabla f(x) \|^2$. This induces a linear speedup in the convergence rate of SGD, and we will show that our algorithm gain the same benefits from parallelization.

We consider the following updates run in parallel by $W$ workers for parameter $0 < k \leq d$, and $k$-contraction operator $\text{sparse}_k: \mathbb{R}^d \rightarrow \mathbb{R}^d$ (cf. Definition 6):
\[
x_{t+1} := x_t - \frac{1}{W} \sum_{w=1}^{W} g_{i_w}^w,
\] (3.49)
\[
g_{i_w}^w := \text{sparse}_k(m_{i_w}^w + \eta_t \nabla f_{i_w}(x_t)),
\] (3.50)
\[
m_{i+1}^w := m_t^w + \eta_t \nabla f_{i_w}(x_t) - g_{i_w}^w,
\] (3.51)
where $i_w \sim \text{u.a.r.} [n]$, $m_0^w = 0$ for $w \in [W]$ and $\{\eta_t\}_{t \geq 0}$ denotes a sequence of stepizes. Each worker has its own local memory where it accumulates the non-shared gradients, the shared gradients are averaged over all machines and applied together every round.

We will follow the same proof as in section 3.4 with some slight differences to encompass the multi workers set up and prove a linear speed up in $W$ with a new constraint on $a$, the learning rate shift. The new result is expressed in theorem 9.

Perturbed iterate. The virtual sequence becomes
\[
\tilde{x}_0 := x_0, \quad \tilde{x}_{t+1} := \tilde{x}_t - \frac{1}{W} \sum_{w=1}^{W} \eta_t \nabla f_{i_w}(x_t)
\] (3.52)
and we observe that the difference between the virtual sequence $\tilde{x}_t$ and the true sequence $x_t$ is the average of each worker’s memory,
\[
x_t - \tilde{x}_t = \left( x_0 - \frac{1}{W} \sum_{w=1}^{W} g_{i_w}^w \right) - \left( x_0 - \frac{1}{W} \sum_{w=1}^{W} \eta_t \nabla f_{i_w}(x_t) \right) = \frac{1}{W} \sum_{w=1}^{W} m_{i_w}^w := \bar{m}_t.
\] (3.53)

Bounding the error. To take into account the drop in variance when $W$ workers collaborate, we rewrite corollary 4 into the next Lemma. Observe that the only differences are the $1/W$ factor in the variance term and the $(1 - 2L\eta_t)^{-1}$ factor.
Lemma 8. Let \( \{x_t\}_{t \geq 0} \) and \( \{\tilde{x}_t\}_{t \geq 0} \) be defined as in (3.49) and (3.52), let \( W \geq 1 \) and let \( f_t \) be \( L \)-smooth, \( f \) be \( \mu \)-strongly convex with \( \mathbb{E}_t \|\nabla f_t(x_t) - \nabla f(x_t)\|^2 \leq \sigma^2 \) and \( \bar{m}_t \) as in (3.53). Then

\[
e_t \leq \frac{1}{\eta_t (1 - 2L\eta_t)} \left( 1 - \frac{\eta_t \mu}{2} \right) a_t - a_{t+1} + \eta_t^2 \frac{\sigma^2}{W} + \eta_t (\mu + 2L) \mathbb{E} \|\bar{m}_t\|^2 \tag{3.54}
\]

where \( e_t := \mathbb{E} f(x_t) - f^* \) and \( a_t := \mathbb{E} \|\tilde{x}_t - x^\star\|^2 \).

Proof. We modify slightly the proof of lemma 3 to make \( \sigma^2 \) appear,

\[
\|\tilde{x}_{t+1} - x^\star\|^2 = \|\tilde{x}_t - \frac{\eta_t}{W} \sum_{w=1}^W \nabla f_t^w(x_t) - x^\star\|^2 \tag{3.55}
\]

Taking the expectation with regard to \( I_t = \{i^w_t, w \in [W]\} \), the last term disappears because the average stochastic gradient estimate is unbiased. The second term gives

\[
\eta_t^2 \mathbb{E}_{I_t} \left\| \nabla f(x_t) - \frac{1}{W} \sum_{w=1}^W \nabla f_t^w(x_t) \right\|^2 \leq \eta_t^2 \frac{\sigma^2}{W}, \tag{3.58}
\]

and the first term of (3.57) gives,

\[
\|\tilde{x}_t - x^\star - \eta_t \nabla f(x_t)\|^2 = \|\tilde{x}_t - x^\star\|^2 + \eta_t^2 \|\nabla f(x_t)\|^2 - 2 \eta_t \langle \tilde{x}_t - x^\star, \nabla f(x_t) \rangle \tag{3.59}
\]

where we used that \( \|\nabla f(x_t)\|^2 \leq 2L(f(x_t) - f^*) \) (remark 13). We obtain

\[
\mathbb{E}_{I_t} \|\tilde{x}_{t+1} - x^\star\|^2 \leq \|\tilde{x}_t - x^\star\|^2 + \eta_t^2 \frac{\sigma^2}{W} + 2L \eta_t^2 (f(x_t) - f^*) \tag{3.61}
\]

Bounding the memory. We now have to bound the average distributed memory \( \|\bar{m}_t\|^2 \) as in lemma 3 for the single process case. Observe that

\[
\mathbb{E} \|\bar{m}_t\|^2 = \mathbb{E} \left\| \frac{1}{W} \sum_{w=1}^W m_t^w \right\|^2 \leq \frac{1}{W} \sum_{w=1}^W \mathbb{E} \|m_t^w\|^2 = \mathbb{E} \|m_t\|^2, \tag{3.62}
\]

which implies that the bound on the memory norm for a single process also holds in the parallel setting.

Towards linear speed up. We can finally restate the main theorem (theorem 2) with \( W \) parallel workers. We would like to highlight that the differences are (i) an extra condition on \( a \), (ii) an extra factor 2 in the constant and (iii) the improved \( \sigma^2/W \) term.
Theorem 9. Let $f_i$ be $L$-smooth, $f$ be $\mu$-strongly convex, $0 < k \leq d$, $E_i \|\nabla f_i(x_i)\|^2 \leq G^2$ and $E_i \|\nabla f_i(x_i) - \nabla f(x_i)\|^2 \leq \sigma^2$ for $t = 0, \ldots, T - 1$, where \{x_i\}$_{i \geq 0}$ are generated according to (3.49) for $W \geq 1$ workers, stepsizes $\eta_t = \frac{\alpha}{\mu + t}$ and shift parameter $\alpha > 1$ with $\kappa = L/\mu$. Then for $\alpha > 4$ such that $\max \left(32\kappa, \frac{(\alpha + 1)\frac{d}{\mu} + \rho}{\rho + 1}\right) \leq a$, with $\rho = \frac{\alpha}{(\alpha - 1)(\alpha + 1)^2}$, it holds

$$E f(x_T) - f^* \leq \frac{2T(T + 2a)\alpha^2}{\mu S_T} + \frac{\mu a^3}{4S_T} \|x_0 - x^*\|^2 + \frac{128T(1 + 2\kappa)}{\mu S_T} \left(\frac{4\alpha}{\alpha - 4}\right) \frac{d^2}{k^2} G^2,$$  

(3.63)

where $x_T = \frac{1}{S_T} \sum_{t=0}^{T-1} w_t x_t$, for $w_t = (a + t/2)^2$, and $S_T = \sum_{t=0}^{T-1} w_t \geq \frac{T}{2} T^3$.

Proof. Same as proof of theorem 2 but starting from lemma 8 and using the extra assumption $a \geq 32\kappa$, which lets us replace the new factor of eq. (3.54) by a constant,

$$(1 - 2L\eta_t)^{-1} \leq (1 - 2L\eta_0)^{-1} \leq 2$$

(3.64)

3.6 Discussion

The sparsified SGD with memory algorithm has an extra hyper parameter $a$, the learning rate shift. It is used to adjust the initial learning rate $\eta_0 = 8/\mu a$, to avoid a too high learning rate for the first $k$ steps, which would fill the memory with very high magnitude gradients and the delayed updates would hurt the convergence (see chapter 4). In fact, for any shift $a > 1$ there is a parameter $a(a) > 4$ such that (3.8) holds. However, for the choice $a = O(1)$ one has to set $\alpha$ such that $\frac{2}{\alpha - 4} = \Omega\left(\frac{d}{\mu}\right)$ and the last term in (3.8) will be of order $O\left(\frac{d^3}{\mu T^2}\right)$, thus requiring $T = \Omega\left(\frac{d^{1.5}}{\mu}\right)$ steps to yield convergence. For $\alpha \geq 5$ we have $\frac{2}{\alpha - 4} = O(1)$ and the last term is only of order $O\left(\frac{d^2}{\mu T^3}\right)$ instead. However, increasing $\alpha$ requires typically a large shift $a$. Observe

$$\frac{(\alpha + 1)\frac{d}{\mu} + \rho}{\rho + 1} \leq 1 + (\alpha + 1)\frac{d}{k} \leq (\alpha + 2)\frac{d}{k},$$

(3.65)

that is setting $a = (\alpha + 2)\frac{d}{k}$ is enough. We like to stress that in general it is not advisable to set $a \gg (\alpha + 2)\frac{d}{k}$ as the first two terms in (3.8) depend on $a$. In practice, it often suffices to set $a = \frac{d}{k}$, as we will discuss in chapter 4. The following remark gives a simplified version of the rate for advisable parameter $a$.

Remark 10. Setting $\alpha = 5$ and $a = (\alpha + 2)\frac{d}{k}$ satisfy theorem 3 conditions. With this choice, equation (3.8) simplifies to

$$E f(x_T) - f^* \leq \frac{G^2}{\mu T} + O\left(\frac{d^2 G^2 \kappa}{\mu T^2}\right) + O\left(\frac{d^2 G^2}{\mu T^3}\right),$$

(3.66)

for $\kappa = \frac{T}{\mu}$. We observe that for $T = \Omega\left(\frac{d^{1.5}}{\mu}\right)$ the first term is dominating, and Algorithm 3 converges at rate $O\left(\frac{G^2}{\mu T}\right)$, the same rate as vanilla SGD (Lacoste-Julien et al. 2012).

To estimate the second term in (3.8) we used the property $E \mu \|x_0 - x^*\| \leq 2G$ for $\mu$-strongly convex $f$, as derived in (Rakhlin et al. 2012, Lemma 2).

In the parallel setting, we get a linear speed up on the main term $W$ parallel workers. Unfortunately, the assumption that $a$ is of order $O(\kappa)$ is a bit stronger. The dependency on $\alpha$ of the first and second terms of eq. (3.63) will yield some terms $O(\kappa/T^2)$ and $O(\kappa^3/T^3)$. Some future work on this initial learning rate could also improve the rate of (Sprech 2018) where $\kappa$ appears as well.

Finally, it is interesting to note that our proof does not require the sparsification operator to be unbiased, but only to satisfy the contraction property (definition 5). The proof also works for $k < 1$, where we would retrieve a coordinate only every $k^{-1}$ updates on average, necessarily not deterministically to satisfy the contraction property at every iteration. This ultra sparsification scheme would fall back to increasing the mini-batch size of each machine.

26
Chapter 4

Experiments

We implemented the sparsified SGD with memory algorithm to evaluate its convergence properties in practice. We publish two implementations: a simulation framework in Python and a more efficient code in C++. In this chapter, we present numerical experiments to illustrate the excellent convergence properties and communication efficiency of Mem-SGD. As the usefulness of the scheme has already been shown in practical applications (Dryden et al., 2016; Aji and Heafield, 2017; Lin et al., 2018), we focus here on a few particular aspects.

First, we compare different sparsification operators and their impact on the memory. We also verify the impact of the initial learning rate that did come up in the statement of Theorem 2. We then compare our method with QSGD (Alistarh et al., 2017) (section 2.4.1) which decreases the communication cost in SGD by using random quantization operators but without memory. We finally evaluate the hypothesis that the sparse updates of Mem-SGD allow a better parallelization than asynchronous parallel SGD (i.e. HOGWILD! (Niu et al., 2011)) on dense problems.

4.1 Implementations

We implement vanilla SGD and sparsified SGD to solve binary classification problems using logistic regression. As seen in section 1.3.1, the associated objective function is

\[ \mathcal{L}(x, A, b) = \sum_{i=1}^{n} \log(1 + \exp(-b_i a_i^\top x)) + \frac{\lambda}{2} \|x\|^2 \] (4.1)

where \( a_i \in \mathbb{R}^d \) and \( b_i \in \{-1, +1\} \) are the data samples, and we employ a standard \( L2 \)-regularizer. The regularization parameter is set to \( \lambda = 1/n \) for both datasets following Schmidt et al. (2017). Our learning rate is \( \eta_t = \gamma(\lambda(t + a))^{-1} \).

Hardware. Experiments were run on an Ubuntu 16.04 machine with a 24 cores processor Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz.

4.1.1 Python implementation (simulation)

We used the numpy library (Oliphant, 2006) in Python to run our experiments. This high-level implementation is not optimized for speed per iteration but for readability and simplicity. We only report convergence per iteration and relative speedups because unequal efforts have been made to speed up the different implementations.

The main challenge with Python involves multiprocessing and shared memory. By default Python duplicates numpy arrays to be accessed from multiple processes. It raises two issues: on the one hand, processes cannot update concurrently a single parameter \( x \) in shared memory and do not collaborate, on the other hand, the data matrices are replicated overflowing the memory. Our solution is to share low-level C arrays between processes and re-wrap them with numpy arrays as demonstrated in code 7 (Appendix B).
4.1.2 C++ implementation

We recoded our algorithm in C++17 using Eigen (Guennebaud et al., 2010), a template library for linear algebra. This implementation follows two objectives: (i) improve the speed of our sparsified SGD implementation in Python and (ii) evaluate how efficient is the scikit-learn implementation of SGD for Logistic Regression using Cython. Using a lower level programming language simplifies the interactions between processes and limits the overhead. We also use the OpenMP library to parallelize our code on multiple cores and evaluate the speed up induced by the increasing number of cores. The OpenMP one-liner parallelization is shown in code 2.

4.1.3 About sparsity

We found that the main factor of optimization lies in leveraging the sparsity of the dataset. Bottou (2012) presents a trick to keep the gradient update sparse (if the sample \( \mathbf{a}_i \) is sparse) by splitting the parameter \( \mathbf{w} \) into \( \mathbf{w}' \) and a scalar \( s \), which enable rescaling—hence regularization—in constant time. This trick speeds up our vanilla SGD implement by an order of magnitude on sparse datasets (see table 4.2).

Adding the stochastic gradient \( \nabla f_i(\mathbf{x}_t) \) (with regularized loss) into the memory \( \mathbf{m}_t \) involves a dense update on vector of size \( d \), which is particularly expensive for big \( d \). To avoid this dense operation, we can evaluate the accumulated regularization term only when a given coordinate is applied and use the memory for the main part of the loss only. The update for a coordinate \( j \) picked (or not) by the sparse \( k \) operator from the memory becomes

\[
x_{t+1,j} = \begin{cases} 
  x_{t,j} \left( 1 - \lambda \sum_{l=1}^{t} \eta_l \right) - (\mathbf{m}_t + \eta_t \nabla f_i(\mathbf{x}_t))_j & \text{if coordinate } j \text{ is retrieved} \\
  x_{t,j} & \text{otherwise}
\end{cases}
\]

(4.2)

where \( \{u_{t,j}\}_{t \geq 0, j \in [d]} \) contains the last time the \( j \)th coordinate was updated and is initialized to 0 for all \( j \). These new update equations only involve sparse vector additions and scalar multiplications. It yields the same speed up as the sparse trick on vanilla SGD. Note that this implementation details becomes incorrect in a multiprocess setting because the assumption that \( x_{t,j} \) remains unchanged when \( j \) is not picked by sparse \( k \) is wrong. It also makes top\(_k\) only compare the main loss without regularization.

4.1.4 Python wrapping

We wrapped our C++ code with Python interface to ease the experimentation process. It is a well-documented recipe and we show a sample of the code needed to make the interface between C++ and Python in code 3.

4.2 Datasets

We evaluate sparsified SGD Memory on binary classification tasks. We consider the following two datasets from UCI Machine Learning repository (Dheeru and Karra Taniskidou, 2017) whose statistics are listed in table 4.1 below.

- **epsilon** (Sonnenburg et al., 2008): a dense dataset of \( n = 400'000 \) samples and \( d = 2'000 \) features released for the PASCAL Challenge 2008. The features are normalized to \( N(0,1) \) and each sample vector is normalized to a unit vector.

- **RCV1** (Lewis et al., 2004): the Reuters Corpus Volume I dataset gathers newswire stories, each represented by a \( 47'236 \) sparse feature vector. Each sample is a rescaled word count vector computed using the Cornell ltc term weighting (Buckley et al., 1994) of the article word occurrences, a form of TF-idf score. We train on the test dataset as it is bigger and more relevant for larger scale experiment (677'399 > 20'242).
### 4.3 Baselines

We compare our implementations to the *SGDClassifier* optimizer in scikit-learn, it is a widely used both in academia and in industry. In all the experiments, we plot a dotted black line at the loss returned by this algorithm to spot if our optimizer has not converged close to the optimum.

The scikit-learn SGD classifier has been heavily optimized and is much faster than our Python code, but our implementation is intended for research, choosing extendability and readability over speed. The speed comparison on the two datasets is presented in Table 4.2 below. Our C++ version exhibits only a slight slow down compared to scikit-learn. We want to highlight that sparse SGD with memory is as fast as SGD on RCV1 using the sparsification trick, however it is slower on the dense dataset *epsilon* and it is not clear why. We could speed up Avg SGD using the trick from Bottou (2012) to keep the average estimate only using only sparse operations.

#### Table 4.1: Datasets statistics.

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$d$</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>400/000</td>
<td>2000</td>
<td>100%</td>
</tr>
<tr>
<td>RCV1-test</td>
<td>677/399</td>
<td>47/236</td>
<td>0.15%</td>
</tr>
</tbody>
</table>

Table 4.2: Time measured in seconds for 1 epoch on 3 runs. Average computes the quadratic weighted mean estimate online. Sparse Mem-SGD uses the trick in section 4.1.3.

<table>
<thead>
<tr>
<th></th>
<th>SGD</th>
<th>Avg SGD</th>
<th>Sparse Mem-SGD</th>
<th>Mem-SGD</th>
<th>Avg Mem-SGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCV1-test</td>
<td>sklearn</td>
<td>0.76 ± 0.09</td>
<td>0.86 ± 0.13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>0.95 ± 0.01</td>
<td>11.25 ± 0.07</td>
<td>0.96 ± 0.02</td>
<td>10.85 ± 0.21</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>371.72 ± 2.96</td>
<td>478.01 ± 25.67</td>
<td>-</td>
<td>1051.86 ± 4.64</td>
</tr>
<tr>
<td>epsilon</td>
<td>sklearn</td>
<td>0.06 ± 0.05</td>
<td>0.73 ± 0.01</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>0.97 ± 0.07</td>
<td>11.22 ± 0.13</td>
<td>10.85 ± 0.20</td>
<td>9.96 ± 0.21</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>104.44 ± 0.14</td>
<td>119.09 ± 3.04</td>
<td>-</td>
<td>171.16 ± 3.07</td>
</tr>
</tbody>
</table>

#### Table 4.2: Time measured in seconds for 1 epoch on 3 runs. Average computes the quadratic weighted mean estimate online. Sparse Mem-SGD uses the trick in section 4.1.3.

### 4.4 Results

We present the results of our experiments in this section. We focus on the results of the C++ implementation as it is the current state of our research. Results of Python experiments are presented in Appendix C.

#### 4.4.1 Convergence illustration

We study the convergence of sparsified SGD with memory for different sparsification operators and different values of $k$. We try to illustrate the convergence property shown in the theory part. We use the stepsize $\eta_t = (\lambda(t+a))^{-1}$ with hyper-parameters $\gamma = 1$ and $a = 10d$. We compute the final estimate $\tilde{x}$ as a weighted average of all iterates $x_t$ with weights $w_t = (t+a)^2$ as indicated by Theorem 2. We use $k \in \{1,2,3\}$ for *epsilon* and $k \in \{1,5,10\}$ for *RCV1*. The results are depicted in Figure 4.1. The topk variant consistently outperforms randk and stays very close to SGD. The impact of the delay $a$ in the learning rate is crucial: setting it to 10 instead of order $O(d/k)$ dramatically hurts the memory and requires time to recover from the high initial learning rate as you can see in Figure 4.1 (labeled “no delay”). In the zoomed figure 4.2, we distinguish that top1 has a slight advantage over SGD for *epsilon*, maybe because some coordinate are very noisy and should not be followed at the beginning with a high learning rate.

We also analyze the evolution of the norm of the memory and its highest magnitude coordinate during the training (fig. 4.3). As expected, topk maintains a smaller memory norm than randk and operators with better contraction property (randk vs rand1) exhibit smaller memory norm which helps to converge faster. Some coordinates are updated more often than others with the topk sparsifier. The distribution of number of times each coordinate is picked, shown in fig. 4.4 is skewed. For RCV1-test, many coordinates are almost never updated while the distribution exhibits a heavy queue (cut on
Figure 4.1: Convergence of Mem-SGD using different sparsification operators compared to full SGD with quadratic weighting \( w_t = (t + a)^2 \) and theoretical learning \( \eta_t = (\lambda(t + a))^{-1} \).

Figure 4.2: Zoom of fig. 4.1
the figure) having some coordinates picked \( > 40'000 \) times. It contrasts with the \( \text{rand}_k \) which picks coordinate uniformly and has a normal distribution around 50. Interestingly, the \( \text{top}_k \) coordinate selected distribution is skewed the other way around for \( \epsilon \text{psilon} \) (dense dataset), which is what we expect from \( \text{top}_k \), focusing only on a few coordinates. Which let us conclude that the skew for \( \text{RCV1} \) comes from the sparsity of the dataset features itself.

### 4.4.2 Comparison with QSGD

We experimentally verified the convergence properties of \( \text{Mem-SGD} \) for different sparsification operators and stepsizes but we want to further evaluate its fundamental benefits in terms of sparsity enforcement and reduction of the communication bottleneck. The gain in communication cost of SGD with memory is very high for dense datasets—using the \( \text{top}_1 \) strategy on \( \epsilon \text{psilon} \) dataset improves the amount of communication by \( 10^3 \) compared to SGD. For the sparse dataset, SGD can readily use the sparsity of the gradients. The effective dimension of the gradients on \( \text{RCV1} \) is around \( 47'236 \times 0.15\% \approx 71 \), which lets SGD use sparse gradients and in expectation share only 142 floating-point numbers per update. Nevertheless, the improvement for \( \text{top}_{10} \) on \( \text{RCV1} \) is of approximately an order of magnitude.

Now we compare \( \text{Mem-SGD} \) with the QSGD compression scheme [Alistarh et al. (2017)] which reduces communication cost by random quantization (using Python implementation). The accuracy (and the compression ratio) in QSGD is controlled by a parameter \( s \), corresponding to the number.
of quantization levels. Ideally, we would like to set the quantization precision in QSGD such that the number of bits transmitted by QSGD and Mem-SGD is identical. However, even for the lowest precision, QSGD needs to send the sign and the index of $O(\sqrt{d})$ coordinates. It is therefore not possible to reach the compression level of sparsification operators that only transmit a constant number of bits per iteration. Hence, we did not enforce this condition and resorted to picking reasonable levels of quantization in QSGD ($s = 2^b$ with $b \in \{2, 4, 8\}$). Figure 4.5 shows that Mem-SGD with top$_1$ and top$_{10}$ on epsilon and RCV1 converges as fast as QSGD in term of iterations for 8 and 4-bits respectively. Note that $b$-bits stands for the number of bits used to encode $s = 2^b$ levels but the actual number of bits transmitted in QSGD can be reduced using Elias coding. According to Alistarh et al. (2017, Theorem 3.2) QSGD with 8-bits levels needs to share $10^5$ bits per update on epsilon, and 4-bits levels on RCV1 needs $10^3$ bits per update. As shown in the bottom of Figure 4.5 we are transmitting two orders of magnitude fewer bits with the top$_1$ sparsifier for epsilon and one order of magnitude for RCV1, concluding that sparsification offers a much more aggressive and performant strategy than quantization.

4.4.3 Multicore scale up

We implement a parallelized version of Mem-SGD, as described in algorithm 4. The enforced sparsity allows us to do the update in shared memory using a lock-free mechanism as in Niu et al. (2011). For this experiment, we evaluate the final iterate $x_T$ instead of the weighted average $\bar{x}_T$ above. We kept the same learning rate as above.
Algorithm 4 Parallel-Mem-SGD

1: Initialize shared variable $x$ and $m_w^0 = 0, \forall w \in [W]$
2: parallel for $w$ in 1 ... $W$ do
3: for $t$ in 0 ... $T - 1$ do
4: Sample $i^w_t$ uniformly in $[n]$
5: $g^w_t \leftarrow \text{sparse}_{i^w_t}(m^w_t + \eta_t \nabla f_{i^w_t}(x))$
6: $x = x - g^w_t$ \hspace{1cm} ▷ update in shared memory
7: $m^w_{t+1} \leftarrow m^w_t + \eta_t \nabla f_{i^w_t}(x) - g^w_t$
8: end for
9: end parallel for

Figures 4.6 and 4.7 show the speed-up obtained when we increase the number of cores for each dataset. The experiment is run on a single machine, hence no inter-node communication is used. The colored area depicts the best and worst results of 3 independent runs for each dataset.

We first notice that rand100 does not converge for more than 15 cores on RCV1-test and also requires more epoch to converge on epsilon. This can be due to the replication of the memory on each worker, which hurts a lot when the norm of each memory is high. We tried to divide the updates by $W$ to the update as proposed in our extension to multiple workers (section 3.5), thus taking the mean of the memories which reduces the “global” memory norm. It made the updates too small and did not yield enough progress. However, top1 exhibit a very good scaling property on both datasets, consistently requiring a constant number of epochs to converge and benefiting a lot from parallelization for the sorting part, which makes the relative speedup look better (for RCV1-test) even though the
wall-clock time is similar. Two directions of improvement are (i) to approximate the top \( k \) selection to speed up the sparsification as in (Lin et al., 2018), or (ii) to optimize more complex models where the sparsification time becomes negligible compared to the time to compute the gradient.

For sparse datasets (e.g. RCV1-test) leveraging the sparsity of the input and thus of the gradient is crucial to be efficient. The difference is visible on RCV1-test with the gap in wall-clock time between rand\(_{100}\) and sgd. It is not clear if using the sparse memory update in eq. (4.2) would hurt when parallelized as we are not applying a proper regularization term, our experiment did not give us exploitable results.

It is still not clear from these experiments if the sparse updates enforced by the sparse\(_ k \) operator improve the asynchronous parallel implementation. The speed-up is especially remarkable for classic SGD and HOGWILD! [Niu et al., 2011] on the dense \( \epsilon \) dataset, as the dense gradient updates would usually imply many update conflicts for (i.e. hurting convergence because of gradients computed on stale iterates) or require the use of locking (i.e. hurting speed). We do not outperform HOGWILD! but top\(_1\) is on par with it and in a distributed setting, the win of communication would be high.

4.5 Discussion

It is difficult to compare SGD with Mem-SGD as many overheads can be introduced by other programming factors, especially in Python. We tried to be as fair as possible with QSGD but it can be argued that we did not invest enough time searching the best possible convergence set up (learning rate and quantization levels) to make a fair comparison. Our goal was to exhibit that sparsification is a much more aggressive compression scheme and should be used if it does not hurt convergence. In practice, we believe that tuning the hyper-parameters (learning rate) of sparsified SGD with memory is less tedious and is a common task one can do on a subset of the dataset.

We could also benefit from memory clipping to avoid the initial explosion of the memory and make our method more stable, however, we decided to stay close to what we propose in theory and not follow the road of alchemy. Even though we do not have a proof for non-averaged sparsified SGD with memory, it worked very well in practice. This is crucial for parallelism as keeping an online estimate of the average adds some cooperation between the workers.

Sparsified SGD with memory is simple to parallelize and does not involve a big overhead compared to SGD. To harvest its full benefits, it should be applied to bigger models than generalized linear models, for instance neural networks, to leverage multi-device architecture and benefit from the impressive communication savings of this algorithm.
Conclusion

To the best of our knowledge, we provide the first concise convergence analysis of sparsified SGD for smooth and convex functions in sequential and parallel setting. This extremely communication-efficient variant of SGD enforces sparsity of the applied updates by only updating a constant number of coordinates in every iteration. This way, the method overcomes the communication bottleneck of SGD, while still enjoying the same convergence rate in terms of stochastic gradient computations. The extension of the proof for sparsified SGD with parallel workers support the efficient use of distributed systems in practice. The asynchronous setting would be the next step to use asynchronous sparsified SGD on multiple cores with shared memory or with the parameter server architecture.

Our experiments verify the drastic reduction in communication cost by demonstrating that MEM-SGD requires one to two orders of magnitude fewer bits to be communicated than QSGD [Alistarh et al. (2017)] while converging to the same accuracy. The experiments show an advantage for the top-k sparsification over random sparsification, the memory norm being kept much smaller which helps convergence as seen in theory. We observe that in the multi-core shared memory implementation, having distributed memories hurt the convergence compared to a simple shared memory implementation that just writes the dense updates in lock-free asynchronous fashion (like Hogwild! [Niu et al. (2011)]).

Despite the fact that our analysis does not yet comprise the asynchronous setting, we feel that this is the domain where sparsified SGD might have the largest impact. It has already been shown in practice that gradient sparsification can be efficiently applied to bandwidth memory limited systems such as multi-GPU training for neural networks [Dryden et al. (2016); Aji and Heafield (2017); Lin et al. (2018); Strom (2015); Seide et al. (2014)]. By enforcing sparsity, the scheme is not only communication efficient, it also becomes more eligible for asynchronous implementations, as limiting sparsity assumptions can be revoked.
Appendices
Appendix A

Useful facts

We gather the tricks that we used in this paper.

**Fact 1.** Given two vectors $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{R}^d$, we have

$$2 \langle \mathbf{a}, \mathbf{b} \rangle \leq \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2.$$  \hfill (A.1)

**Proof.**

$$0 \leq \|\mathbf{a} - \mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2 \langle \mathbf{a}, \mathbf{b} \rangle$$  \hfill (A.2)\[\Box\]

**Fact 2.** Given two vectors $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{R}^d$, we have

$$\|\mathbf{a} + \mathbf{b}\|^2 \leq (1 + \gamma) \|\mathbf{a}\|^2 + (1 + \gamma^{-1}) \|\mathbf{b}\|^2, \forall \gamma > 0.$$  \hfill (A.3)

**Proof.**

$$\|\mathbf{a} + \mathbf{b}\|^2 = \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 + 2 \langle \mathbf{a}, \mathbf{b} \rangle$$  \hfill (A.4)

$$= \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 + 2 \left( \gamma^{1/2} \mathbf{a}, \gamma^{-1/2} \mathbf{b} \right)$$  \hfill (A.5)

$$= (1 + \gamma) \|\mathbf{a}\|^2 + (1 + \gamma^{-1}) \|\mathbf{b}\|^2$$  \hfill (A.6)\[\Box\]

**Fact 3.** Given a sequence of vectors $\mathbf{a}_i \in \mathbb{R}^d$ with $i \in \{1, \ldots, s\}$,

$$\left\| \sum_{i=1}^{s} \mathbf{a}_i \right\|^2 \leq s \sum_{i=1}^{s} \|\mathbf{a}_i\|^2.$$  \hfill (A.7)

**Proof.** The statement is true for $s = 1$. By induction,

$$\left\| \sum_{i=1}^{s} \mathbf{a}_i \right\|^2 = \left\| \sum_{i=1}^{s-1} \mathbf{a}_i + \mathbf{a}_s \right\|^2$$  \hfill (A.8)

$$= \left\| \sum_{i=1}^{s-1} \mathbf{a}_i \right\|^2 + \|\mathbf{a}_s\|^2 + 2 \sum_{i=1}^{s-1} \langle \mathbf{a}_i, \mathbf{a}_s \rangle$$  \hfill (A.9)

$$\leq (s - 1) \sum_{i=1}^{s-1} \|\mathbf{a}_i\|^2 + \|\mathbf{a}_s\|^2 + \sum_{i=1}^{s-1} \|\mathbf{a}_i\|^2 + (s - 1) \|\mathbf{a}_s\|^2$$  \hfill (A.10)

$$= s \sum_{i=1}^{s} \|\mathbf{a}_i\|^2$$  \hfill (A.11)

where we used in (A.10) the induction hypothesis and fact [1]\[\Box\]
Fact 4. Let $\eta_t = \frac{1}{c+t}$, for $c \geq 1$. Then $\eta_t^2 \left(1 - \frac{2}{c}\right) \leq \eta_{t+1}^2$.

Proof. Observe

$$
\eta_t^2 \left(1 - \frac{2}{c}\right) = \frac{c - 2}{c(c + t)^2} \leq \frac{c - 2}{(c + t + 1)^2(c - 2)} = \eta_{t+1}^2.
$$

(A.12)

where the inequality follows from

$$(c + t + 1)^2(c - 2) = c(c + t)^2 + (c - 2)(1 + 2(t + c)) - 2(c + t)^2
\leq -2t^2 - 2ct - 3c^2 - 2 \leq 0
$$

(A.13)

Remark 11. Given $L$-smooth functions $f_i : \mathbb{R}^d \to \mathbb{R}$ for $i \in [N]$, the function $F$ defined such that

$$
F(x) = \frac{1}{N} \sum_{i=1}^{N} f_i(x)
$$

is $L$-smooth.

Proof. From the definition and by linearity of the gradient operator, for $x, y \in \mathbb{R}^d$,

$$
F(y) = \frac{1}{N} \sum_{i=1}^{N} f_i(y)
\leq \frac{1}{N} \sum_{i=1}^{N} \left[ f_i(x) + \langle \nabla f_i(x), y - x \rangle + L \|x - y\|^2 \right]
= F(x) + \langle \nabla F(x), y - x \rangle + L \|x - y\|^2
$$

(A.16)

(A.17)

Remark 12. If a function $f : \mathbb{R}^d \to \mathbb{R}$ is $L$-smooth it satisfies

$$
f(x) \geq f(y) + \langle \nabla f(y), x - y \rangle + \frac{1}{2L} \|\nabla f(x) - \nabla f(y)\|^2, \forall x, y \in \mathbb{R}^d.
$$

(A.18)

Proof. We define $g(x) := f(x) - x^T \nabla f(y)$ which is convex with global minimum at $y$. Then,

$$
g(y) \leq g \left( x - \frac{1}{L} \nabla g(x) \right)
\leq g(x) + \nabla g(x)^T \left( -\frac{1}{L} \nabla g(x) \right) + \frac{L}{2} \|\nabla g(x)\|^2
\leq g(x) - \frac{1}{2L} \|\nabla g(x)\|^2
$$

(A.19)

(A.20)

(A.21)

Then

$$
f(y) - y^T \nabla f(y) \leq f(x) - x^T \nabla f(y) - \frac{1}{2L} \|\nabla f(x) - \nabla f(y)\|^2
$$

(A.22)

Remark 13. If a function $f : \mathbb{R}^d \to \mathbb{R}$ is $L$-smooth it satisfies

$$
\|\nabla f(x)\|^2 \leq 2L(f(x) - f^*), \forall x \in \mathbb{R}^d
$$

(A.23)

Proof. Using remark 12

$$
\|\nabla f(x)\|^2 = \|\nabla f(x) - \nabla f(x_*)\|^2 \leq 2L(f(x) - f(x_*)) = 2L(f(x) - f(x_*))
$$

(A.24)
Appendix B

Code snippets

Code 1 Share numpy array with multiple processes.

```
import numpy as np
import multiprocessing as multiprocessing

# load dataset in X, y and initialize weights in w

# unwrap numpy arrays into C arrays
XRaw = mp.sharedctypes.RawArray('d', X)
yRaw = mp.sharedctypes.RawArray('d', y)
wRaw = mp.sharedctypes.RawArray('d', w)

def _train(XRaw, yRaw, wRaw, shape):
    # re wrap the C arrays with numpy array
    X = np.ctypeslib.as_array(XRaw)
    X.shape = shape
    y = np.ctypeslib.as_array(yRaw)
    y.shape = (shape[0],)
    w = np.ctypeslib.as_array(wRaw)
    w.shape = (shape[1],)

    # start training...

# pass the C arrays to the new process
process = mp.Process(
    target=_train,
    args=(XRaw, yRaw, wRaw, X.shape, (numSamples, numFeatures)))

# start the process
process.start()
process.join()
```
Code 2 Multiprocessing with C++ OpenMP library.

```c++
#include <omp.h>

omp_set_dynamic(0);
omp_set_num_threads(num_cores);

#pragma omp parallel for firstprivate(memory, dw, t, lr, sampleIndex, queue, rho) 
    shared(w, wEstimate, stopFlag) schedule(dynamic) collapse(2)
for (size_t epoch = 0; epoch < numEpochs; epoch++)
{
    for (size_t iteration = 0; iteration < numSamples; iteration++)
    {
        // ...
    }
}
```
from libcpp.string cimport string
from libcpp cimport bool

// skipped definition vector, Results, Dataset, PyDataset

cdef extern from "logisticSGD.h":
c
cdef class PygisticSGD:
c
cdef class PygisticSGD:
c
cdef _cinit_(self, numEpochs, lrType, lr, tau, lambda_,
weightingScheme, useMemory, takeK, takeTop, cores, printPerEpoch):
self._thisptr = new LogisticSGD(numEpochs, lrType.encode(), lr, tau, lambda_,
weightingScheme.encode(), useMemory, takeK, takeTop, cores, printPerEpoch)
c
cdef __dealloc__(self):
del self._thisptr

cdef _init_(self, numEpochs, lrType, lr, tau, lambda_, weightingScheme, u
seMemory, takeK, takeTop, cores, printPerEpoch):
pas

cdef fit(self, PyDataset dataset, float until=0):
assert dataset.is_loaded()
res = self._thisptr.fit(dataset._thisptr, until)
return self._result_to_tuple(res)
c
cdef loss(self, PyDataset dataset):
assert dataset.is_loaded()
return self._thisptr.loss(dataset._thisptr)
c
cdef accuracy(self, PyDataset dataset):
assert dataset.is_loaded()
return self._thisptr.accuracy(dataset._thisptr)
c
cdef _result_to_tuple(self, Result* res):
fields = [res.ts, res.losses, res.timers, res.memoryNorm, res.memoryMax,
res.selectedCoordCount]
results = [[] for _ in range(len(fields))]
for field, r in zip(fields, results):
    for elem in field:
        r.append(elem)
return results
Appendix C

Figures

C.1 Python experiments

C.1.1 Theoretical convergence

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>$\frac{\gamma}{a}$ $\frac{2}{d/k}$</td>
</tr>
<tr>
<td>RCV1-test</td>
<td>$\frac{\gamma}{a}$ $\frac{2}{10d/k}$</td>
</tr>
</tbody>
</table>

Table C.1: Learning rate $\eta = \gamma/(\lambda(t + a))$.

Figure C.1: Convergence of Mem-SGD using different sparsification operators compared to full SGD with quadratic weighting $w_t \equiv (t + a)^2$ and theoretical learning rates (parameters in Table C.1).

C.1.2 Multicore experiment

Multicore experiments were first simulated in Python. Figure C.2 shows the relative speedup of the different algorithms. The behavior shown in the figure was not exactly reproduced in C++ and it might be due to some overhead in Python. We used constant $\eta_t \equiv 0.05$ for epsilon and reused the parameters from Table C.1 for RCV1.
Figure C.2: Multicore CPU time speed up comparison between Mem-SGD and lock-free SGD.
Bibliography


Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017. URL http://archive


Alexander Rakhlin, Ohad Shamir, and Karthik Sridharan. Making gradient descent optimal for


Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In Haizhou Li, Helen M. Meng, Bin Ma, Engsiong Chng, and Lei Xie, editors, INTERSPEECH, pages 1058–1062. ISCA, 2014. URL http://dblp.uni-trier.de/db/conf/interspeech/interspeech2014.html#SeideFDLY14


