Sparsified SGD with Memory

Sebastian U. Stich  Jean-Baptiste Cordonnier  Martin Jaggi

Machine Learning and Optimization Laboratory (MLO)
EPFL, Switzerland

Abstract

Huge scale machine learning problems are nowadays tackled by distributed optimization algorithms, i.e. algorithms that leverage the compute power of many devices for training. The communication overhead is a key bottleneck that hinders perfect scalability. Various recent works proposed to use quantization or sparsification techniques to reduce the amount of data that needs to be communicated, for instance by only sending the most significant entries of the stochastic gradient (top-$k$ sparsification). Whilst such schemes showed very promising performance in practice, they have eluded theoretical analysis so far.

In this work we analyze Stochastic Gradient Descent (SGD) with $k$-sparsification or compression (for instance top-$k$ or random-$k$) and show that this scheme converges at the same rate as vanilla SGD when equipped with error compensation (keeping track of accumulated errors in memory). That is, communication can be reduced by a factor of the dimension of the problem (sometimes even more) whilst still converging at the same rate. We present numerical experiments to illustrate the theoretical findings and the good scalability for distributed applications.

1 Introduction

Stochastic Gradient Descent (SGD) [29] and variants thereof (e.g. [10, 16]) are among the most popular optimization algorithms in machine- and deep-learning [5]. SGD consists of iterations of the form

$$x_{t+1} := x_t - \eta_t g_t,$$

for iterates $x_t, x_{t+1} \in \mathbb{R}^d$, stepsize (or learning rate) $\eta_t > 0$, and stochastic gradient $g_t$ with the property $\mathbb{E}[g_t] = \nabla f(x_t)$, for a loss function $f : \mathbb{R}^d \to \mathbb{R}$. SGD addresses the computational bottleneck of full gradient descent, as the stochastic gradients can in general be computed much more efficiently than a full gradient $\nabla f(x_t)$. However, note that in general both $g_t$ and $\nabla f(x_t)$ are dense vectors of size $d$, i.e. SGD does not address the communication bottleneck of gradient descent, which occurs as a roadblock both in distributed as well as parallel training. In the setting of distributed training, communicating the stochastic gradients to the other workers is a major limiting factor for many large scale (deep) learning applications, see e.g. [3, 21, 33, 44]. The same bottleneck can also appear for parallel training, e.g. in the increasingly common setting of a single multi-core machine or device, where locking and bandwidth of memory read operations for the common shared parameter $x_t$ often forms the main bottleneck, see e.g. [14, 18, 25].

A remedy to address these issues seems to enforce applying smaller and more efficient updates $\text{comp}(g_t)$ instead of $g_t$, where $\text{comp} : \mathbb{R}^d \to \mathbb{R}^d$ generates a compression of the gradient, such as by lossy quantization or sparsification. We discuss different schemes below. However, too

---

1 Note that the stochastic gradients $g_t$ are dense vectors for the setting of training neural networks. The $g_t$ themselves can be sparse for generalized linear models under the additional assumption that the data is sparse.
aggressive compression can hurt the performance, unless it is implemented in a clever way: 1Bit-SGD \cite{1BitSGD, 1BitSGD2} combines gradient quantization with an error compensation technique, which is a memory or feedback mechanism. We in this work leverage this key mechanism but apply it within more the more general setting of SGD. We will now sketch how the algorithm uses feedback of to correct for errors accumulated in previous iterations. Roughly speaking, the method keeps track of a memory vector \( \mathbf{m} \) which contains the sum of the information that has been suppressed thus far, i.e. \( \mathbf{m}_{t+1} := \mathbf{m}_t + \mathbf{g}_t - \text{comp}(\mathbf{g}_t) \), and injects this information back in the next iteration, by transmitting \( \text{comp}(\mathbf{m}_{t+1} + \mathbf{g}_{t+1}) \) instead of only \( \text{comp}(\mathbf{g}_{t+1}) \). Note that updates of this kind are not unbiased (even if \( \text{comp}(\mathbf{g}_{t+1}) \) would be) and there is also no control over the delay after which the single coordinates are applied. These are some (technical) reasons why there exists no theoretical analysis of this scheme up to now.

In this paper we give a concise convergence rate analysis for SGD with memory and \( k \)-compression operators, such as (but not limited to) top-\( k \) sparsification. Our analysis also supports ultra-sparsification operators for which \( k < 1 \), i.e. where less than one coordinate of the stochastic gradient is applied on average in \( (1) \). We not only provide the first convergence result of this method, but the result also shows that the method converges at the same rate as vanilla SGD.

1.1 Related Work

There are several ways to reduce the communication in SGD. For instance by simply increasing the amount of computation before communication, i.e. by using large mini-batches (see e.g. \cite{large_mini_bachts, large_mini_bachts2}), or by designing communication-efficient schemes \cite{communication_efficient}. These approaches are a bit orthogonal to the methods we consider in this paper, which focus on quantization or sparsification of the gradient.

Several papers consider approaches that limit the number of bits to represent floating point numbers \cite{limit_num_bits, limit_num_bits2, limit_num_bits3}. Recent work proposes adaptive tuning of the compression ratio \cite{adaptive_tuning}. Unbiased quantization operators not only limit the number of bits, but quantize the stochastic gradients in such a way that they are still unbiased estimators of the gradient \cite{unbiased_quantization, unbiased_quantization2}. The ZipML framework also applies this technique to the data \cite{ZipML}. Sparsification methods reduce the number of non-zero entries in the stochastic gradient \cite{sparsification1, sparsification2}.

A very aggressive sparsification method is to keep only very few coordinates of the stochastic gradient by considering only the coordinates with the largest magnitudes \cite{aggressive_sparsification1, aggressive_sparsification2}. In contrast to the unbiased schemes it is clear that such methods can only work by using some kind of error accumulation or feedback procedure, similar to the one the we have already discussed \cite{1BitSGD, 1BitSGD2}, as otherwise certain coordinates could simply never be updated. However, in certain applications no feedback mechanism is needed \cite{no_feedback}. Also more elaborate sparsification schemes have been introduced \cite{more_elaborate_schemes}.

Asynchronous updates provide an alternative solution to disguise the communication overhead to a certain amount \cite{asynchronous_updates}. However, those methods usually rely on a sparsity assumption on the updates \cite{sparsity_assumption1, sparsity_assumption2}, which is not realistic e.g. in deep learning. We like to advocate that combining gradient sparsification with those asynchronous schemes seems to be a promising approach, as it combines the best of both worlds. Other scenarios that could profit from sparsification are heterogeneous systems or specialized hardware, e.g. accelerators \cite{heterogeneous_systems, specialized_hardware}.

Convergence proofs for SGD \cite{convergence_proofs} typically rely on averaging the iterates \cite{averaging1, averaging2, averaging3}, though convergence of the last iterate can also be proven \cite{last_iterate}. For our convergence proof we rely on averaging techniques that give more weight to more recent iterates \cite{perturbed_iterate1, perturbed_iterate2, perturbed_iterate3}, as well as the perturbed iterate framework from Mania et al. \cite{perturbed_iterate_framework} and techniques from \cite{perturbed_iterate_framework2, perturbed_iterate_framework3}.

Simultaneous to our work, \cite{related_schemes1, related_schemes2} at NIPS 2018 propose related schemes. Whilst Tang et al. \cite{related_schemes1} only consider unbiased stochastic compression schemes, Alistarh et al. \cite{related_schemes2} study biased top-\( k \) sparsification. Their scheme also uses a memory vector to compensate for the errors, but their analysis suffers from a slowdown proportional to \( k \), which we can avoid here. Another simultaneous analysis of Wu et al. \cite{related_schemes3} at ICML 2018 is restricted to unbiased gradient compression. This scheme also critically relies on an error compensation technique, but in contrast to our work the analysis is restricted to quadratic functions and the scheme introduces two additional hyperparameters that control the feedback mechanism.

\footnote{See Definition 2.1}
1.2 Contributions

We consider finite-sum convex optimization problems $f : \mathbb{R}^d \to \mathbb{R}$ of the form

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x), \quad x^* := \arg\min_{x \in \mathbb{R}^d} f(x), \quad f^* := f(x^*), \quad (2)$$

where each $f_i$ is $L$-smooth\(^3\) and $f$ is $\mu$-strongly convex\(^4\). We consider a sequential sparsified SGD algorithm with error accumulation technique and prove convergence for $k$-compression operators, $0 < k \leq d$ (for instance the sparsification operators top-$k$ or random-$k$). For appropriately chosen stepsizes and an averaged iterate $\bar{x}_T$ after $T$ steps we show convergence

$$E f(\bar{x}_T) - f^* = O \left( \frac{G^2}{\mu T} \right) + O \left( \frac{\epsilon^2 L G^2 \kappa}{\mu T^2} \right) + O \left( \frac{\epsilon^2 G^2}{\mu T^3} \right), \quad (3)$$

for $\kappa = \frac{L}{\mu}$ and $G^2 \geq E \|\nabla f_i(x_t)\|^2$. Not only is this, to the best of our knowledge, the first convergence result for sparsified SGD with memory, but the result also shows that for $T = \Omega(\frac{d^2}{\kappa})$ the first term is dominating and the convergence rate is the same as for vanilla SGD.

We introduce the method formally in Section 2 and show a sketch of the convergence proof in Section 3. In Section 4 we include a few numerical experiments for illustrative purposes. The experiments highlight that top-$k$ sparsification yields a very effective compression method and does not hurt convergence. We also report results for a parallel multi-core implementation of SGD with memory that show that the algorithm scales as well as asynchronous SGD and drastically decreases the communication cost without sacrificing the rate of convergence. We like to stress that the effectiveness of SGD variants with sparsification techniques has already been demonstrated in practice [1, 9, 21, 33, 37].

Although we do not yet provide convergence guarantees for parallel and asynchronous variants of the scheme, this is the main application of this method. For instance, we like to highlight that asynchronous SGD schemes [2, 25] could profit from the gradient sparsification. To demonstrate this use-case, we include in Section 4 a set of experiments for a multi-core implementation.

2 SGD with Memory

In this section we present the sparsified SGD algorithm with memory. First we introduce sparsification and quantization operators which allow us to drastically reduce the communication cost in comparison with vanilla SGD.

2.1 Compression and Sparsification Operators

We consider compression operators that satisfy the following contraction property:

**Definition 2.1** ($k$-contraction). *For a parameter $0 < k \leq d$, a $k$-contraction operator is a (possibly randomized) operator $\text{comp} : \mathbb{R}^d \to \mathbb{R}^d$ that satisfies the contraction property*

$$E \|x - \text{comp}(x)\|^2 \leq \left( 1 - \frac{k}{d} \right) \|x\|^2, \quad \forall x \in \mathbb{R}^d. \quad (4)$$

The contraction property is sufficient to obtain all mathematical results that are derived in this paper. However, note that (4) does not imply that $\text{comp}(x)$ is a necessarily sparse vector. Also dense vectors can satisfy (4). One of the main goals of this work is to derive communication efficient schemes, thus we are particularly interested in operators that also ensure that $\text{comp}(x)$ can be encoded much more efficiently than the original $x$.

The following two operators are examples of $k$-contraction operators with the additional property of being $k$-sparse vectors:

\(^3 f_i(y) \leq f_i(x) + \langle \nabla f_i(x), y - x \rangle + \frac{L_i}{2} \|y - x\|^2, \quad \forall x, y \in \mathbb{R}^d, \quad i \in [n].\)

\(^4 f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2} \|y - x\|^2, \quad \forall x, y \in \mathbb{R}^d.\)
We note that our setting is more general than simply measuring sparsity in terms of the cardinality, i.e.

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

\[ (\text{rand}_k(x, \omega))_i := \begin{cases} (x)_i, & \text{if } i \in \omega, \\ 0, & \text{otherwise}, \end{cases} \]

where \( \pi \) is a permutation of \([d]\) such that \((x)_i \geq (x)_{\pi(i)}\) for \(i = 1, \ldots, d - 1\). We abbreviate \(\text{rand}_k(x)\) whenever the second argument is chosen uniformly at random, \(\omega \sim \text{u.a.r.} \Omega_k\).

It is easy to see that both operators satisfy Definition 2.1 of being a \(k\)-contraction. For completeness the proof is included in Appendix A.1.

We note that our setting is more general than simply measuring sparsity in terms of the cardinality, i.e. the non-zero elements of vectors in \(\mathbb{R}^d\). Instead, Definition 2.1 can also be considered for quantization or e.g. floating point representation of each entry of the vector. In this setting we would for instance measure sparsity in terms of the number of bits that are needed to encode the vector. By this, we can also use stochastic rounding operators (similar as the ones used in [3], but with different scaling) as compression operators according to [4]. Also gradient dropping [11] trivially satisfies [4], though with different parameter \(k\) in each iteration.

**Remark 2.3 (Ultra-sparsification).** We like to highlight that many other operators do satisfy Definition 2.1 not only the two examples given in Definition 2.2. As a notable variant is to pick a random coordinate of a vector with probability \(\frac{k}{d}\), for \(0 < k < 1\), property [4] holds even if \(k < 1\). I.e., it suffices to transmit on average less than one coordinate per iteration (this would then correspond to a mini-batch update).

### 2.2 Variance Blow-up for Unbiased Updates

Before introducing SGD with memory we first discuss a motivating example. Consider the following variant of SGD, where \((d - k)\) random coordinates of the stochastic gradient are dropped:

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

where \(i \sim \text{u.a.r.} [n]\). It is important to note that the update is unbiased, i.e. \(E g_t = \nabla f(x)\). For carefully chosen step sizes \(\eta_t\) this algorithm converges at rate \(O\left(\frac{\sigma^2}{t}\right)\) on strongly convex and smooth functions \(f\), where \(\sigma^2\) is an upper bound on the variance, see for instance [46]. We have

\[ \sigma^2 = E \left\| \frac{d}{k} \cdot \text{rand}_k(\nabla f_t(x)) - \nabla f(x) \right\|^2 \leq E \left\| \frac{d}{k} \cdot \text{rand}_k(\nabla f_t(x)) \right\|^2 \leq \frac{d}{k} \cdot \sigma^2 \]

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

It is well known that by using mini-batches the variance of the gradient estimator can be reduced. If we consider in (6) the estimator \(g_t := \frac{d}{k} \cdot \text{rand}_k \left( \frac{1}{k} \sum_{i \in I_t} \nabla f_t(x_i) \right)\) for \(\tau = \lceil \frac{k}{d} \rceil\), and \(I_t \sim \text{u.a.r.} \binom{n}{\tau}\), instead, we have

\[ \sigma^2 = E \left\| \frac{d}{k} \cdot \text{rand}_k \left( \frac{1}{\tau} \sum_{i \in I_t} \nabla f_t(x_i) \right) \right\|^2 \leq \frac{d}{k \tau} \cdot \sigma^2 \]

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 increased number of iterations or increased computation. However, this is no longer true when the information in (6) is not dropped, but kept in memory. To illustrate this, assume \(k = 1\) and that index \(i\) has not been selected by the \(\text{rand}_1\) operator in iterations \(t = t_0, \ldots, t_{s-1}\), but is selected in iteration \(t_s\). Then the memory \(m_{t_s} \in \mathbb{R}^d\) contains this past information \(m_{t_s} = \sum_{t=t_0}^{t_s-1} (\nabla f_t(x))_i\). Intuitively, we would expect that the variance of this estimator is now reduced by a factor of \(s\) compared to the naïve estimator in (6), similar to the mini-batch update in (7). Indeed, SGD with memory converges at the same rate as vanilla SGD, as we will demonstrate below.
We consider the following algorithm for parameter $0$.

**Algorithm 1** 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. $g_t \leftarrow \text{comp}_k (m_t + \eta_t \nabla f_{i_t}(x_t))$
   5. $x_{t+1} \leftarrow x_t - g_t$
   6. $m_{t+1} \leftarrow m_t + \eta_t \nabla f_{i_t}(x_t) - g_t$
3. end for

**Algorithm 2** PARALLEL-MEM-SGD

1. Initialize shared variable $x$
2. parallel for $w$ in $1 \ldots W$
   3. for $t$ in $0 \ldots T - 1$
   4. Sample $i^w_t$ uniformly in $[n]$
   5. $g^w_t \leftarrow \text{comp}_k (m^w_t + \eta_t \nabla f_{i^w_t}(x))$
   6. $x \leftarrow x - g^w_t$
   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

Figure 1: Left: The MEM-SGD algorithm. Right: Implementation for multi-core experiments.

### 2.3 SGD with Memory: Algorithm and Convergence Results

We consider the following algorithm for parameter $0 < k \leq d$, using a compression operator $\text{comp}_k : \mathbb{R}^d \rightarrow \mathbb{R}^d$ which is a $k$-contraction (Definition 2.1)

$$
\begin{align*}
    x_{t+1} &:= x_t - g_t, & g_t &:= \text{comp}_k (m_t + \eta_t \nabla f_i(x_t)), & m_{t+1} &:= m_t + \eta_t \nabla f_{i_t}(x_t) - g_t, & (8)
\end{align*}
$$

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 [1] Note that the gradients get multiplied with the stepsize $\eta_t$ at the timestep $t$ when they put into memory, and not when they are (partially) retrieved from the memory.

We state the precise convergence result for Algorithm [1] in Theorem 2.4 below. In Remark 2.6 we give a simplified statement in big-O notation for a specific choice of the stepsizes $\eta_t$.

**Theorem 2.4.** Let $f_i$ be $L$-smooth, $f$ be $\mu$-strongly convex, $0 < k \leq d$, $E_t \|\nabla f_i(x_t)\|^2 \leq G^2$ for $t = 0, \ldots, T - 1$, where $\{x_t\}_{t \geq 0}$ are generated according to (8) for stepsizes $\eta_t = \frac{\mu}{8(k+1)}$ and shift parameter $\alpha > 1$. Then for $\alpha > 4$ such that $\frac{(\alpha+1)^2+\rho}{\rho+1} \leq \alpha$, with $\rho := \frac{4\alpha}{(\alpha-4)(\alpha+1)^2}$, it holds

$$
E f(x_T) - f^* \leq \frac{4T(T + 2\alpha)}{\mu S_T} G^2 + \frac{\mu^2}{8S_T} \|x_0 - x^*\|^2 + \frac{64T(1 + 2\frac{\alpha}{\rho})}{\mu S_T} \left(\frac{4\alpha}{\alpha-4}\right) \frac{d^2}{k^2} G^2, \tag{9}
$$

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

**Remark 2.5 (Choice of the shift $\alpha$).** Theorem 2.4 says that for any shift $\alpha > 1$ there is a parameter $\alpha(\alpha) > 4$ such that (9) holds. However, for the choice $\alpha = O(1)$ one has to set $\alpha$ such that $\frac{\alpha}{\alpha-4} = \Omega(\frac{2}{\kappa})$ and the last term in (9) will be of order $O\left(\frac{d^3}{k^2 T^2}\right)$, thus requiring $T = \Omega\left(\frac{d^6}{k^2 \kappa^2}\right)$ steps to yield convergence. For $\alpha \geq 5$ we have $\frac{\alpha}{\alpha-4} = O(1)$ and the last term is only of order $O\left(\frac{d^2}{k^2 T^2}\right)$ instead. However, this requires typically a large shift. Observe $\frac{(\alpha+1)^2+\rho}{\rho+1} \leq 1 + (\alpha+1)^2 \leq (\alpha+2)^2$, i.e. setting $\alpha = (\alpha+2)^\frac{1}{2}$ is enough. We like to stress that in general it is not advisable to set $\alpha \geq (\alpha+2)^\frac{1}{2}$ as the first two terms in (9) depend on $\alpha$. In practice, it often suffices to set $\alpha = \frac{d}{k}$, as we will discuss in Section 2.

**Remark 2.6.** As discussed in Remark 2.5 above, setting $\alpha = 5$ and $\alpha = (\alpha+2)^\frac{1}{2}$ is feasible. With this choice, equation (9) simplifies to

$$
E f(x_T) - f^* \leq O\left(\frac{G^2}{\mu T}\right) + O\left(\frac{\mu^2 G^2}{\mu T^2}\right) + O\left(\frac{\mu^2 G^2}{\mu T^3}\right), \tag{10}
$$

for $\kappa = \frac{1}{\mu}$. To estimate the second term in (10) we used the property $E \mu \|x_0 - x^*\| \leq 2G$ for $\mu$-strongly convex $f$, as derived in [28, Lemma 2]. We observe that for $T = \Omega\left(\frac{d^6}{k^{1/2}}\right)$ the first term is dominating, and Algorithm [1] converges at rate $O\left(\frac{G^2}{\mu T}\right)$, the same rate as vanilla SGD [17].

### 3 Proof Outline

We now give an outline of the proof. The proofs of the lemmas are given in Appendix A.2.
**Perturbed iterate analysis.** Inspired by the perturbed iterate framework in [22] and [18] we first define a virtual sequence \(\{\tilde{x}_i\}_{i\geq0}\) in the following way:

\[
\tilde{x}_0 = x_0, \quad \tilde{x}_{i+1} = \tilde{x}_i - \eta_i \nabla f_i(x_i),
\]

where the sequences \(\{x_i\}_{i\geq0}, \{\eta_i\}_{i\geq0}\) and \(\{t_i\}_{i\geq0}\) are the same as in [8]. Notice that

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

**Lemma 3.1.** Let \(\{x_i\}_{i\geq0}\) and \(\{\tilde{x}_i\}_{i\geq0}\) be defined as in (8) and (11) and let \(f_i\) be \(L\)-smooth and \(f\) be \(\mu\)-strongly convex with \(\mathbb{E}_t \|\nabla f_i(x_i)\|^2 \leq G^2\). Then

\[
\mathbb{E} \|\tilde{x}_{i+1} - x^*\|^2 \leq \left( 1 - \frac{\eta_s \mu^k}{2} \right) \mathbb{E} \|\tilde{x}_i - x^*\|^2 + \eta_i^2 G^2 - \eta_i e_t + \eta_i (\mu + 2L) \mathbb{E} m_t^2,
\]

where \(e_t := \mathbb{E} f(x_t) - f^*\).

**Bounding the memory.** From equation (13) it becomes clear that we should derive an upper bound on \(\mathbb{E} \|m_t\|^2\). For this we will use the contraction property (4) of the compression operators.

**Lemma 3.2.** Let \(\{x_i\}_{i\geq0}\) as defined in (8) for \(0 < k \leq d\), \(\mathbb{E}_i \|\nabla f_i(x_i)\|^2 \leq G^2\) and stepsizes \(\eta_t = \frac{8}{\mu(a+t)}\) with \(a, \alpha > 4\), as in Theorem 2.4. Then

\[
\mathbb{E} \|m_t\|^2 \leq \eta_t^2 \frac{4a}{\alpha - 4} \frac{d^2}{k^2} G^2.
\]

**Optimal averaging.** Similar as discussed in [17, 28, 34] we have to define a suitable averaging scheme for the iterates \(\{x_i\}_{i\geq0}\) to get the optimal convergence rate. In contrast to [17] that use linearly increasing weights, we use quadratically increasing weights, as for instance [34, 56].

**Lemma 3.3.** Let \(\{a_t\}_{t\geq0}, \{e_t\}_{t\geq0}, e_t \geq 0\), be sequences satisfying

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

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

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

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

**Proof of Theorem 2.4.** The proof of the theorem immediately follows from the three lemmas that we have presented in this section and convexity of \(f\), i.e. we have \(\mathbb{E} f(\tilde{x}_T) - f^* \leq \frac{1}{S_T} \sum_{t=0}^{T-1} w_t e_t\) in (16), for constants \(A = G^2\) and \(B = (\mu + 2L) \frac{4a}{\alpha - 4} \frac{d^2}{k^2} G^2\).

## 4 Experiments

We present numerical experiments to illustrate the excellent convergence properties and communication efficiency of Mem-SGD. As the usefulness of SGD with sparsification techniques has already been shown in practical applications [11, 21, 33, 37] we focus here on a few particular aspects. First, we verify the impact of the initial learning rate that did come up in the statement of Theorem 2.4. We then compare our method with QSGD [5] which decreases the communication cost in SGD by using random quantization operators, but without memory. Finally, we show the performance of the parallel SGD depicted in Algorithm 2 in a multi-core setting with shared memory and compare the speed-up to asynchronous SGD.

### 4.1 Experimental Setup

**Models.** The experiments focus on the performance of Mem-SGD applied to logistic regression. The associated objective function is \(\frac{1}{n} \sum_{i=1}^n \log(1 + \exp(-b_i a_i^\top x)) + \frac{1}{2} \|x\|^2\), 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 [32].
Datasets. We consider a dense dataset, epsilon [35], as well as a sparse dataset, RCV1 [20] where we train on the larger test set. Statistics on the datasets are listed in Table 1 below:

<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>2'000</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 1: Datasets statistics.

Implementation. We use Python3 and the numpy library [15]. Our code is open-source and publicly available at [github.com/epfml/sparsifiedSGD](https://github.com/epfml/sparsifiedSGD). We emphasize that our high level implementation is not optimized for speed per iteration but for readability and simplicity. We only report convergence per iteration and relative speedups, but not wall-clock time because unequal efforts have been made to speed up the different implementations. Plots additionally show the baseline computed with the standard optimizer LogisticSGD of scikit-learn [26]. Experiments were run on an Ubuntu 18.04 machine with a 24 cores processor Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz.

4.2 Verifying the Theory

We study the convergence of the method using the stepsizes $\eta_t = \gamma/(\lambda(t+a))$ and hyperparameters $\gamma$ and $a$ set as in Table 2. We compute the final estimate $\bar{x}$ as a weighted average of all iterates $x_t$ with weights $w_t = (t+a)^2$ as indicated by Theorem [2.4]. The results are depicted in Figure 2. We use $k \in \{1, 2, 3\}$ for epsilon and $k \in \{10, 20, 30\}$ for RCV1 to increase the difference with large number of features. The top$k$ variant consistently outperforms rand$k$ and sometimes outperforms vanilla SGD, which is surprising and might come from feature characteristics of the datasets. We also evaluate the impact of the delay $a$ in the learning rate: setting it to 1 instead of order $O(d/k)$ dramatically hurts the memory and requires time to recover from the high initial learning rate (labeled “without delay” in Figure 2).

We experimentally verified the convergence properties of 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 top1 strategy on epsilon dataset improves the amount of communication by $10^3$ compared to SGD. For the sparse dataset, SGD can readily use the given sparsity of the gradients. Nevertheless, the improvement for top10 on RCV1 is of approximately an order of magnitude.

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

Table 2: Learning rate $\eta_t = \gamma/(\lambda(t+a))$. 

![Figure 2: Convergence of Mem-SGD using different sparsification operators compared to full SGD with theoretical learning rates (parameters in Table 2).](image-url)
4.3 Comparison with QSGD

Now we compare MEM-SGD with the QSGD compression scheme [3] which reduces communication cost by random quantization. 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 are identical and compare their convergence properties. However, even for the lowest precision, QSGD needs to send the sign and index of \( O(\sqrt{d}) \) coordinates. It is therefore not possible to reach the compression level of sparsification operators such as top-\( k \) or random-\( k \), that only transmit a constant number of bits per iteration (up to logarithmic factors). Hence, we did not enforce this condition and resorted to pick reasonable levels of quantization in QSGD (\( s = 2^b \) with \( b \in \{2, 4, 8\} \)). Note that \( b \)-bits stands for the number of bits used to encode \( s = 2^b \) levels but the number of bits transmitted in QSGD can be reduced using Elias coding. As a fair comparison in practice, we chose a standard learning rate \( \gamma_0/(1 + \gamma_0 \lambda t)^{-1} \) [6], tuned the hyperparameter \( \gamma_0 \) on a subset of each dataset (see Appendix B). Figure 3 shows that MEM-SGD with top\( _1 \) on epsilon and RCV1 converges as fast as QSGD in term of iterations for 8 and 4-bits. As shown in the bottom of Figure 3, we are transmitting two orders of magnitude fewer bits with the top\( _1 \) sparsifier concluding that sparsification offers a much more aggressive and performant strategy than quantization.

4.4 Multicore experiment

We implement a parallelized version of MEM-SGD, as depicted in Algorithm 2. The enforced sparsity allows us to do the update in shared memory using a lock-free mechanism as in [25]. For this experiment we evaluate the final iterate \( x_T \) instead of the weighted average \( \bar{x}_T \) above, and use the learning rate \( \eta_t \equiv (1 + t)^{-1} \).

Figure 4 shows the speed-up obtained when increasing the number of cores. We see that both sparsified SGD and vanilla SGD have a linear speed-up, the slopes are dependent of the implementation details. But we observe that PARALLEL-MEM-SGD with a reasonable sparsification parameter \( k \) does not suffer of having multiple independent memories. The experiment is run on a single machine with a

---

8 Encoding the indices of the top-\( k \) or random-\( k \) elements can be done with additional \( O(k \log d) \) bits. Note that \( \log d \leq 32 \leq \sqrt{d} \) for both our examples.
24 core processor, hence no inter-node communication is used. The main advantage of our method—overcoming the communication bottleneck—would be even more visible in a multi-node setup. In this asynchronous setup, SGD with memory computes gradients on stale iterates that differ only by a few coordinates. It encounters fewer inconsistent read/write operations than lock free asynchronous SGD and exhibits better scaling properties on the RCV1 dataset. The top\(_k\) operator performs better than rand\(_k\) in the sequential setup, but this is not the case in the parallel setup.

5 Conclusion

We provide the first concise convergence analysis of sparsified SGD [1, 9, 33, 37]. 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.

Our experiments verify the drastic reduction in communication cost by demonstrating that MEM-SGD requires one to two orders of magnitude less bits to be communicated than QSGD [3] while converging to the same accuracy. The experiments show an advantage for the top-\(k\) sparsification over random sparsification in the serial setting, but not in the multi-core shared memory implementation. There, both schemes are on par, and show better scaling than a simple shared memory implementation that just writes the unquantized updates in a lock-free asynchronous fashion (like Hogwild! [25]).

The theoretical insights to MEM-SGD that were developed here should facilitate the analysis of the same scheme in the parallel (as developped in [8]) and the distributed setting. 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 [1, 9, 21, 33, 37]. By delivering sparsity no matter if the original gradients were sparse or not, our scheme is not only communication efficient, but becomes more eligible for asynchronous implementations as well. While those were so far limited by strict sparsity assumptions (as e.g. in [25]), our approach might make such methods much more widely applicable.

Acknowledgments

We would like to thank Dan Alistarh for insightful discussions in the early stages of this project and Frederik Künstner for his useful comments on the various drafts of this manuscript. We acknowledge funding from SNSF grant 200021_175796, Microsoft Research JRC project ‘Coltrain’, as well as a Google Focused Research Award.
References


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, Eungsiong Chng, and Lei Xie, editors, INTERSPEECH, pages 1058–1062. ISCA, 2014.


