Optimal Distributed Online Prediction

Ofer Dekel OFERD@MICROSOFT.COM
Ran Gilad-Bachrach RANG@MICROSOFT.COM
Ohad Shamir OHADSH@MICROSOFT.COM
Lin Xiao LIN.XIAO@MICROSOFT.COM
Microsoft Research, USA

Abstract

Online prediction methods are typically studied as serial algorithms running on a single processor. In this paper, we present the *distributed mini-batch* (DMB) framework, a method of converting a serial gradient-based online algorithm into a distributed algorithm, and prove an asymptotically optimal regret bound for smooth convex loss functions and stochastic examples. Our analysis explicitly takes into account communication latencies between computing nodes in a network. We also present robust variants, which are resilient to failures and node heterogeneity in an asynchronous distributed environment. Our method can also be used for distributed stochastic optimization, attaining an asymptotically linear speedup. Finally, we empirically demonstrate the merits of our approach on large-scale online prediction problems.

1. Introduction

Many natural prediction tasks can be cast as stochastic online prediction problems. These are often discussed in the serial setting, where the computation takes place on a single processor. However, when the examples arrive at a high rate and have to be processed in real time, there may be no choice but to distribute the computation across multiple cores or multiple cluster nodes. For example, modern search engines process thousands of queries a second, and indeed they are implemented as distributed algorithms that run in massive data-centers. In this paper, we focus on such large-scale and high-rate online prediction problems, where parallel and distributed computing is critical to providing a real-time service.

Appearing in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.

First, we begin by defining the stochastic *online prediction* problem. Suppose that we observe a stream of examples $z_1, z_2, \ldots$, where each z_i is sampled independently from a fixed unknown distribution over a sample space $\mathcal{Z}$. Before observing each z_i, we predict a point w_i from a set W. After making the prediction w_i, we observe z_i and suffer the loss $f(w_i, z_i)$, where f is a predefined loss function. Then we use z_i to improve our prediction mechanism for the future (e.g., using a stochastic gradient method). We measure the quality of our predictions using the notion of *regret*, defined as

$$R(m) \;=\; \sum_{i=1}^{m} \left(f(w_i, z_i) - f(w^\star, z_i) \right) \;,$$

where $w^\star$ is the minimizer of $F(w) = \mathbb{E}_z \left[f(w, z) \right]$ over W. In this paper, we restrict our discussion to convex prediction problems, where f is convex in its first argument and W is a closed convex subset of $\mathbb{R}^n$. Also, for brevity we focus on bounds on the expected regret, but note that our results are readily extendable to high-probability bounds on the actual regret.

We model our distributed computing system as a set of k *nodes*, each being an independent processor, and a *network* that enables the nodes to communicate with each other. Each node receives and handles an incoming stream of examples from an outside source, such as a load balancer/splitter, and simultaneously may communicate with neighboring nodes in the network. As in the real world, we assume that the network has a limited bandwidth, so the nodes cannot simply share all of their information, and messages sent over the network incur a non-negligible latency.

How well can we perform in such a distributed environment? At one extreme, an ideal (but unrealistic) solution is to run a serial algorithm on a single "super" processor that is k times faster than a standard node. This solution is optimal, simply because any distributed algorithm can be simulated on a fast-enough single processor. Using gradient-based serial algorithms, the regret bound in this case is $O(\sqrt{m})$, and is known to be

optimal (Nemirovski & Yudin, 1983; Abernethy et al., 2009). At the other extreme, a trivial solution to our problem is to have each node run an independent copy of a serial algorithm, without any communication over the network. However, the regret of this solution scales poorly with the network size k. Assuming that each node processes m/k examples, the total regret is $O(k\sqrt{m/k}) = O(\sqrt{km})$ - namely, a factor of $\sqrt{k}$ worse than the ideal solution. The first sanity-check that any distributed online prediction algorithm must pass is that it outperforms this trivial solution.

In this paper, we present the *distributed mini-batch* (DMB) framework, which has the following important properties:

- It can use any gradient-based update rule for serial online prediction as a black box, and convert it into a distributed online prediction algorithm.
- If the loss function $f(w, z)$ is smooth in w (i.e. has a Lipschitz-continuous gradient), then our method attains an asymptotically optimal regret bound of $O(\sqrt{m})$. Moreover, the coefficient of the dominant term $\sqrt{m}$ is *the same* as in the serial bound, and *independent* of the network size k and other network characteristics.
- When adapted to stochastic gradient based algorithms for stochastic optimization, the DMB algorithm achieves nearly linear speed-up with parallel computing.

Our paper is organized as follows. First, we give a brief discussion of related work in Sec. 1.1, which puts our main results mentioned above in a clear perspective. In Sec. 2, we present our algorithm in a synchronous distributed setup, where all nodes work properly and at the same rate, and the network has no failures. Since these assumptions may not be realistic in all distributed settings, we also develop variants of our method in Sec. 3 and Sec. 4, which are provably robust to heterogeneous nodes, various failures, and changes in the network topology. We conclude in Sec. 5 by presenting several experiments on large-scale prediction problems, which demonstrate the merits of our approach. Due to limitation of space, proofs can be found in Dekel et al. (2010a) and Dekel et al. (2010b).

1.1. Related Work

The problem of distributing online prediction and optimization, including the case of gradient-based algorithms, has received much attention, with notable early references being Tsitsiklis et al. (1986) and Bertsekas & Tsitsiklis (1989). More recent works include Ram et al. (2009); Nedić & Ozdaglar (2009); Zinkevich et al. (2010); Duchi et al. (2010). Despite the large activity in this area, the existing guarantees for all the algorithms we are aware of are either purely asymptotic (without rate of convergence in the optimization setting), or are not significantly better than the trivial solution that has an $O(\sqrt{km})$ regret bound, which scales poorly with the network size.

Perhaps the closest work to ours is Langford et al. (2009), which address distributed online prediction and attempts to show improved $O(\sqrt{m})$ regret guarantees with k parallel workers, compared to the $O(\sqrt{km})$ of the trivial solution. However, the architecture they propose is not scalable beyond a multi-core or small cluster setting. Moreover, due to an error in the proof of Lemma 1 therein, their improved regret bounds only hold in special cases, when $W = \mathbb{R}^n$ yet boundedness of the gradients and compactness of the domain can still be implicitly guaranteed.[1]

As the name *distributed mini-batch* suggests, our approach makes use of the idea of mini-batches, which is not new and has been previously explored in both the serial and parallel setting (see, e.g., Delalleau & Bengio, 2007; Gimpel et al., 2010). However, as far as we know, our work is the first to use this idea in order to obtain such provably strong results in a distributed learning setting.

2. The Distributed Mini-Batch (DMB) Framework

We focus on gradient-based algorithms, that conform to the following template: after each example z_j is processed, the gradient $g_j = \nabla_w f(w_j, z_j)$ is computed, and is used to obtain the next predictor w_{j+1}. The update is performed using a generic *update rule* ϕ:

$$(w_{j+1}, a_{j+1}) = \phi(a_j, g_j, \alpha_j),$$

where a_j is an auxiliary state vector that summarizes necessary past information, and α_j is an iteration-dependent parameter such as a stepsize. This template corresponds to large families of online algorithms, such as mirror descent algorithms (Nemirovski et al., 2009) and dual averaging (Nesterov, 2009; Xiao, 2010). As an example, a special case of the mirror descent algorithm is the well-known online projected gradient descent method of Zinkevich (2003),

$$w_{j+1} = \pi_W\left(w_j - g_j/\alpha_j\right),$$

where π_W is Euclidean projection onto the set W.

[1] See Dekel et al. (2010b) for more details.

2.1. Regret Bounds based on Gradient Variance

Typical regret bounds for the online algorithms mentioned above are based on the norm of the gradients, $\sup_{w\in W,z\in\mathcal{Z}} \|\nabla_w f(w,z)\|$. Our approach is based on a novel theoretical observation, that for stochastic inputs and smooth loss functions, one can prove regret bounds that depend on the *variance* of the stochastic gradients. More precisely, we assume

- *Smoothness:* f is L-smooth in its first argument. More formally, $\forall z \in \mathcal{Z}$ and $\forall w, w' \in W$,

$$\|\nabla_w f(w,z) - \nabla_w f(w',z)\| \leq L\|w - w'\| \ .$$

- *Bounded Gradient Variance:* There exists a constant $\sigma > 0$ such that

$$\forall w \in W,\ \mathbb{E}_z[\|\nabla_w f(w,z) - \nabla_w E_z[f(w,z)]\|^2] \leq \sigma^2.$$

The following theorem exemplifies this for projected gradient descent:

Theorem 1. *Let f be an L-smooth convex loss function, Assume that the stochastic gradient $\nabla_w f(w,z)$ has σ^2-bounded variance for all $w \in W$. Also, let $D = \sqrt{\max_{u,v\in W} \|u-v\|_2^2/2}$. Then by running projected gradient descent on m i.i.d. examples with an appropriate step size, the expected regret is at most*

$$(F(w_1) - F(w^\star)) + D^2L + 2D\sigma\sqrt{m}.$$

See Dekel et al. (2010b) for the proof, as well as appropriate versions for mirror descent and dual averaging algorithms. In either case, if $\nabla F(w^\star) = 0$ (which is the case if $w^\star$ is strictly inside W), then the expected regret bounds can be simplified to

$$\mathbb{E}[R(m)] \ \leq \ 2D^2L + 2D\sigma\sqrt{m} \ . \qquad (1)$$

For brevity, we will use $\psi(\sigma^2, m)$ to denote such variance bounds for predicting on m examples, and will often focus on the case where they are equal to Eq. (1).

2.2. The DMB algorithm

The key observation from the above theoretical results is the following: suppose that instead of updating our predictor after each gradient, we accumulate a *mini-batch* of b gradients with respect to the same predictor, and only then update the predictor based on the average of these b gradients. These averaged gradients have the same expectation as the original gradients, but smaller variance due to the averaging, leading to an improvement in the variance-based regret bounds above. In a nutshell, the DMB framework that we propose uses the distributed network in order to rapidly accumulate gradients with respect to the same fixed predictor w. Once a mini-batch of sufficiently many gradients are accumulated (parameterized by b), the nodes collectively perform a vector-sum operation across the network, which allows each node to obtain the average of these b gradients. This average is then used to update the predictor, using any gradient-based online update rule as a black box. The pseudo-code for any single node appears as Algorithm 1.

Algorithm 1 Distributed mini-batch (DMB)

```
for j = 1, 2, ... do
  initialize ĝ_j := 0
  for s = 1, ..., b/k do
    predict w_j
    receive example z sampled i.i.d. from unknown distribution
    suffer loss f(w_j, z)
    compute g := ∇_w f(w_j, z)
    ĝ_j := ĝ_j + g
  end for
  call the distributed vector-sum to compute the sum of ĝ_j across all nodes
  receive μ/k additional examples and continue predicting using w_j
  finish vector-sum and compute average gradient ḡ_j by dividing the sum by b
  set (w_{j+1}, a_{j+1}) = φ(a_j, ḡ_j, α_j)
end for
```

The regret analysis for this algorithm is based on a parameter μ, which bounds the number of examples processed by the system (all k nodes) during the vector-sum operation. The gradients for these μ examples are not used for updating the predictor. While μ depends on the network structure and communication latencies, it does not scale with the total number of examples m processed by the system. Formally, the regret guarantee is as follows:

Theorem 2. *Let f be an L-smooth convex loss function and assume that the stochastic gradient $\nabla_w f(w,z_i)$ has σ^2-bounded variance for all $w \in W$. If the online update rule has the serial regret bound $\psi(\sigma^2, m)$, then the expected regret of the DMB algorithm over m examples is at most*

$$(b+\mu)\,\psi\left(\frac{\sigma^2}{b}, \left\lceil \frac{m}{b+\mu} \right\rceil\right) \ .$$

Specifically, if $\psi(\sigma^2, m) = 2D^2L + 2D\sigma\sqrt{m}$, and the batch size is chosen to be $b = m^\rho$ for any $\rho \in (0, 1/2)$, the expected regret is $2D\sigma\sqrt{m} + o(\sqrt{m})$. In particular, if $m = b^{1/3}$, the regret is bound by $2D\sigma\sqrt{m} + O(m^{1/3})$.

Note that for serial regret bounds of the form $2D^2L + 2D\sigma\sqrt{m}$, we indeed get an *identical* leading term in the regret bound for the DMB algorithm, implying its asymptotic optimality.

2.3. Speeding up Stochastic Approximation

The DMB framework and its analysis can be easily extended to stochastic optimization. It is well-known that all the online update rules discussed above can be easily converted to stochastic optimization algorithms, by running them over the training set $z_1, \ldots, z_m$, accumulating the sequence of predictors $w_1, w_2, \ldots$, and returning their average $\bar{w}_m$. Moreover, if the expected regret bound of the algorithm is $\psi(\sigma^2, m)$, then the expected optimality gap of the algorithm in a stochastic optimization setting (namely, $\mathbb{E}[F(\bar{w}_m) - \inf_{w \in W} F(w)]$) is at most $\psi(\sigma^2, m)/m$. Therefore, Thm. 2 implies that in a stochastic optimization setting, the optimality gap is asymptotically unaffected by moving from a serial algorithm, to a distributed algorithm using the DMB framework. However, the distributed algorithm can process examples k times faster than the serial algorithm. Therefore, we get an asymptotically linear speed-up in the distributed algorithm, compared to a serial algorithm. Not only is this bound optimal, it is also the first provable demonstration of a non-trivial speedup in using distributed systems for stochastic optimization.

3. Master-Workers Architecture: MaWo-DMB

The DMB algorithm presented earlier assumes that all nodes are making similar progress. However, even in homogeneous systems, which are designed to support synchronous programs, this is hard to achieve, let alone grid environments in which each node may have different capabilities. In this section, we describe a variant of the DMB algorithm that has several desirable properties: It performs on heterogeneous clusters, whose nodes may have varying processing rates; It can handle dynamic network latencies; and it can be made robust using standard fault tolerance techniques.

To provide these properties, we convert the DMB algorithm to work with a single master and multiple workers. Each of the workers receives examples and processes them at its own pace. Periodically, the worker sends the information it collected, i.e., the sum of gradients, to the master. Once the master has collected sufficiently many gradients, it performs a predictor update and broadcasts the new predictor to the workers. We call this algorithm the *master-worker distributed mini-batches* (MaWo-DMB) algorithm.

In well-connected systems, one possible method to implement the communication between the master and the workers is via a shared database. Using a database, each worker can update the gradients it collected on the database, and check for updates from the master. At the same time, the master can check periodically to see if sufficiently many gradients have accumulated in the database. When there are at least b accumulated gradients, the master performs an update and posts the result in a designated place in the database.

The MaWo-DMB algorithm shares a similar asymptotic behavior as the DMB algorithm (e.g. as discussed in Thm. 2). The proof for the DMB algorithm applies to this algorithm as well, where now μ is the number of examples not used in the computation of the next prediction point. μ can be coarsely bounded, for example, by assuming the examples come at a bounded rate, and there is bounded latency in sending messages between the workers and the master. By picking the batch size b to be large enough, we get asymptotically optimal regret.

3.1. Adding Fault Tolerance

The MaWo-DMB has one potential weakness: if the master fails, the algorithm stops making updates. This is a standard problem in master-worker environments. It can be solved using leader election algorithms such as Gallager et al. (1983) and Malpani et al. (2000). If the workers do not receive any signal from the master for a long period of time, they start a process by which they elect a new leader (master). Some of these algorithms are suited to dynamic networks, where the network can be partitioned and reconnected. Therefore, if the network becomes partitioned, each connected component will have its own master.

Another way to introduce robustness to the MaWo-DMB algorithm is by selecting the master only when an update step is to be made. Assume that there is a central database and all workers update it. At predefined time intervals, each worker locks the record in the database; adds the gradients computed to the sum of gradients reported in the database; and adds the number of gradients to the count of the gradients reported in the database. At this point, the worker checks if the count of gradients exceeds b. If it does not, the worker releases the lock and returns to processing examples. However, if the number of gradients does exceed b, the worker performs the update and broadcasts the new prediction point (using the database) before unlocking the database and becoming a worker again. This simple modification creates a distributed master such that any node in the system can be removed without

significantly affecting the progress of the algorithm. In a sense, we are leveraging on the reliability of the database system.

4. Robust Decentralized Architecture: ADMB

In the previous section, we discussed asynchronous algorithms based on a master-workers architecture. Using off-the-shelf fault tolerance methods, one can design simple and robust variants, capable of coping with dynamic and heterogeneous networks. That being said, this kind of approach also has some limitations. For example, accessing a shared database may not be feasible, and utilizing leader-election algorithms is potentially wasteful, as it requires a single master (and a directed acyclic graph) to be agreed upon before predictor updates can be made. Moreover, choosing a computationally weak or communication-constrained node will have severe repercussions. In terms of performance guarantees, it is hard to come up with explicit time guarantees for these algorithms, and hence the effect on the regret incurred by the system is unclear.

In this section, we describe a robust, fully decentralized and asynchronous version of DMB, which we denote as *asynchronous* DMB or ADMB. We provide a formal analysis, and show that ADMB shares the advantages of DMB in terms of dependence on network size and communication latency.

We assume that communication between nodes takes place along some bounded-degree acyclic graph. In addition, each node has a unique numerical index. We will generally use p to denote a given node's index, and let q denote the index of some neighboring node.

Informally, the algorithm works as follows: each node p receives examples, accumulates gradients with respect to its current predictor (which we shall denote as w_p), and shares these gradients with its neighbors. Gradients are eventually propagated throughout the system, with a mechanism to ensure that in each node, no gradient is accumulated twice, and that the gradients are with respect to the same predictor the node is working with. Whenever a node accumulates a batch of b such gradients, it updates w_p. Note that unlike the MaWo-DMB algorithm, here there is no centralized master node responsible for performing the update. Also, we no longer insist on all nodes sharing the exact same predictor at any given time point. Of course, this can lead to each node using a different predictor, and the system will behave as if the nodes all run in isolation. To prevent this, we add a mechanism, which ensures that if a neighboring node q of node p has a predictor based on more updates, then node p will switch to use that predictor. If the two nodes have different predictors based on the same number of updates, then one of them adopts the other node's predictor using a tie-breaking rule. With this mechanism, the predictor with the most gradient updates is propagated quickly throughout the system, so either everyone starts working with this predictor and share gradients, or an even better predictor is obtained somewhere in the system, and is then quickly propagated in turn. Finally, for technical reasons, the predictions themselves are not made with the current predictor w_p, but rather with a running average $\bar{w}_p$ of predictors computed so far.

4.1. The ADMB Algorithm

We now turn to describe the algorithm formally. The algorithm has two global parameters: b, which (as in the DMB algorithm) is the number of gradients whose average is used to update the predictor; and t, which regulates the communication rate between the nodes. Each node p maintains the following data structures:

- A *node state* $S_p = (w_p, \bar{w}_p, v_p)$, where w_p is the current predictor; $\bar{w}_p$ is the running average of predictors computed so far; and v_p counts how many predictors are averaged in $\bar{w}_p$ (equivalently, the number of updates performed according to the online update rule, in order to obtain w_p).
- A vector g_p and associated counter c_p, which hold the sum of gradients computed from examples serviced by node p.
- For each neighboring node q, a vector g_p^q and associated counter c_p^q, which hold the sum of gradients received from node q.

When a node p is initialized, all the variables discussed above are set to zero, The node then begins the execution of the algorithm. The protocol is composed of executing three event-driven functions: handling a new incoming example (Algorithm 2), sending messages to the node's neighbors every t time–units, where a time–unit is arbitrarily defined (Algorithm 3), and processing an incoming message (Algorithm 4). The functions use a subroutine `update_predictor` (Algorithm 5) to update the node's predictor if needed. For simplicity, we will assume that each of those three functions is executed atomically.

Due to the acyclic structure of the network, no single gradient is ever propagated to the same node twice. Thus, the algorithm indeed works correctly, in the sense that the updates are always performed based on independent gradients. Moreover, the algorithm

Algorithm 2 ADMB: Handle new example

Predict using $\bar{w}_p$
Receive example z, suffer loss and compute gradient $\nabla_w f(w_p, z)$
$g_p := g_p + \nabla_w f(w_p, z)$, $c_p := c_p + 1$
if $c_p + \sum_q c_p^q \geq b$ **then**
  `update_predictor`
end if

Algorithm 3 ADMB: Send Message (Every t Time–Units)

For each neighboring node q', send message $\left(p, S_p, g_p + \sum_{q \neq q'} g_p^q, c_p + \sum_{q \neq q'} c_p^q\right)$

Algorithm 4 ADMB: Process Incoming Message

Let (q, S_q, g, c) be the received message
if $S_q.v_q > v_p$ or ($S_q.v_q = v_p$ and $S_q.w_q \neq w_p$ and $q < p$) **then**
  $S_p := S_q$, $g_p := 0$, $c_p := 0$, $\forall q$ $g_p^q := g$, $c_p^q := c$
else
  if $S_q.w_q = w_p$ **then**
    $g_p^q = g$, $c_p^q = c$
    if $c_p + \sum_q c_p^q \geq b$ **then**
      `update_predictor`
    end if
  end if
end if

Algorithm 5 `update_predictor` Subroutine

Use averaged gradient $\frac{g_p + \sum_q g_p^q}{c_p + \sum_q c_p^q}$ to compute updated predictor w_p
$\bar{w}_p := \frac{v_p}{v_p+1}\bar{w}_p + \frac{1}{v_p+1} w_p$
$v_p := v_p + 1$, $g_p := 0$, $c_p := 0$, $\forall q$ $g_p^q := 0$, $c_p^q := 0$

is well-behaved in terms of traffic volume over the network, since any communication link from node p to node q passes at most 1 message every t time–units, where t is a tunable parameter.

As with the MaWo-DMB algorithm, the ADMB algorithm also has some desirable robustness properties, such as capability of working with heterogeneous nodes, adding/removing nodes, and dealing with communication latencies. Moreover, it is robust to network failures: even if the network is split into two (or more) partitions, it only means we end up with two (or more) networks which implement the algorithm in isolation. The system can continue to run and its output will remain valid.

4.2. Analysis

We now turn to discuss the regret performance of the algorithm. Since we have not specified what happens to examples sent to malfunctioning nodes, we will isolate a set of "well-behaved" nodes, and focus on the regret incurred on the examples sent to these nodes. The underlying assumption is that the system is mostly functional for most of the time, so the large majority of examples are processed by such well-behaved nodes.

To that end, let us focus on a particular set of k' nodes, which form a connected component of the communication framework, with diameter d' (which may scale up to k', depending on the network topology). We will define the nodes as *good* during some time period, if all those nodes implement the ADMB algorithm at a reasonably fast rate. More precisely, we assume that for each node, executing each of the functions defining the ADMB algorithm takes at most one time–unit; The communication latency between two adjacent good nodes is at most one time–unit; And the k' nodes receive at most M examples every time–unit. As to other nodes, we only assume that the messages they send to the good nodes reflect a correct node state, as specified earlier. In particular, they may be arbitrarily slow or even completely unresponsive.

Also, we will define a *good time period* to be a time during which all k' nodes are good for sufficiently long. In particular, we assume that the nodes handled $b + 2(t+2)d'M$ examples overall, and were also good for $(t+2)d'$ time–units prior to that time period. As to other time periods, we will only assume that at least *one* of the k' nodes remained operational and implemented the ADMB algorithm (at an arbitrarily slow rate). With these definitions, we have the following result:

Theorem 3. *Suppose the gradient-based update rule has the serial regret bound $\psi(\sigma^2, m)$, and that for any σ^2, $\frac{1}{m}\psi(\sigma^2, m)$ decreases monotonically in m. Let m be the number of examples handled during a sequence of non-overlapping good time periods. Then the expected regret with respect to these examples is at most*

$$\sum_{j=1}^{\lceil m/\mu \rceil} \frac{\mu}{j}\, \psi\left(\frac{\sigma^2}{b}, j\right),$$

where $\mu = b + 2(t+2)d'M$. Specifically, if $\psi(\sigma^2, m) = 2D^2L + 2D\sigma\sqrt{m}$, then the expected regret bound is

$$4D\sigma\sqrt{\left(1 + \frac{2(t+2)d'M}{b}\right)m} + O((b + td'M)\log(m))$$

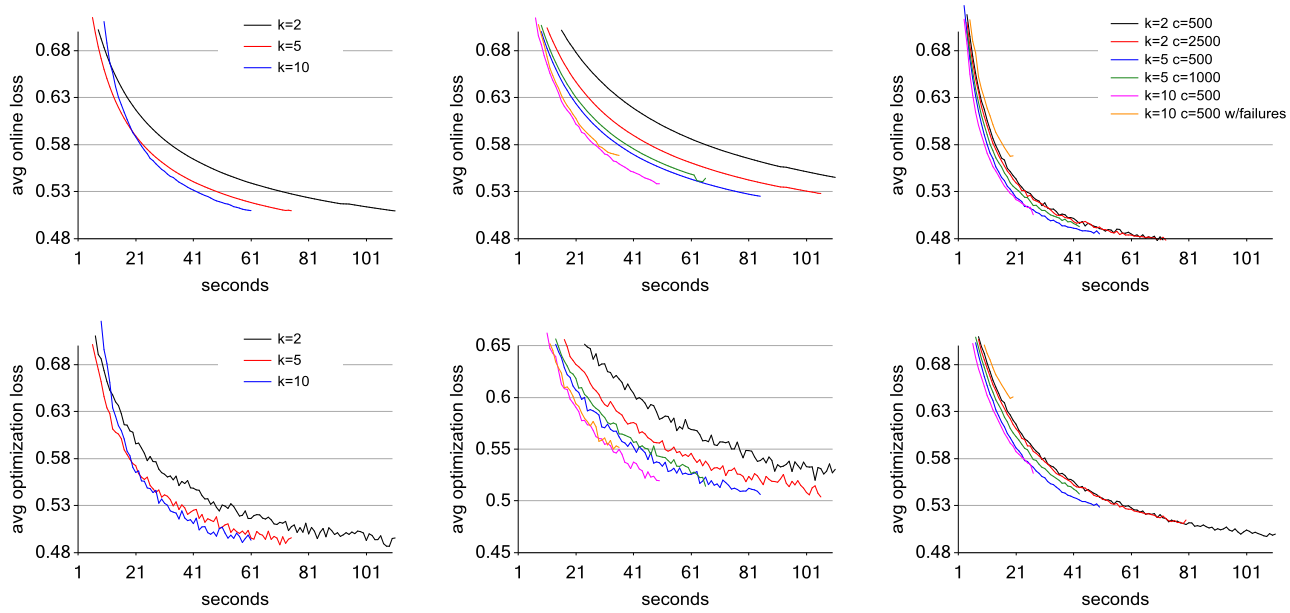

Figure 1. Top: average online loss for DMB (left), MaWo-DMB (center), ADMB(right). Bottom: test loss (on a held-out set) for DMB (left), MaWo-DMB (center), ADMB(right). The mini-batch size in all experiments is $b = 5000$. In each plot, k denotes number of nodes and c denotes chunk size.

We note that t, d', M do not scale with the total number of examples m. Thus, when the batch size b scales as m^ρ for any $\rho \in (0, 1/2)$, we get an asymptotic regret bound of the form $4D\sigma\sqrt{m}+o(\sqrt{m})$. The leading term is virtually the same as the one in the serial regret bound. The only difference is an additional factor of 2, essentially due to the use of averaged predictors.

5. Experiments

We conducted an empirical study using the synchronous distributed mini-batches (DMB) algorithm, the master-worker DMB (MaWo-DMB) algorithm, and the decentralized asynchronous DMB (ADMB) algorithm. We chose the binary classification task of distinguishing between stub-articles and full-articles on Wikipedia. The English version of Wikipedia currently has 5.8 million articles. Each article is associated with a set of categories, and some of the categories are considered to be stub categories. We classified an article as a stub if it is associated with at least one stub category.

We represented each Wikipedia article by a 234-dimensional binary feature vector, constructed as follows: we represented each article by 18 integer features (e.g. article length, title length, number of paragraph, number of links, etc.), and converted each integer feature to 13 binary features, by chunking the integer range to 13 parts on a logarithmic scale.

Our goal was to learn a 234 dimensional linear classifier by minimizing the log-loss function. If w is the current linear classifier and $z = (x, y)$ is an feature vector x and a binary label y, then the log-loss attained by w on z is defined as

$$f(w, z) \;=\; \log\left(1 + \exp(-y\langle w, x\rangle)\right) \;\;.$$

Note that the log-loss is a smooth convex function.

We ran all of our experiments on a small 10-node cluster. The 10 cluster nodes communicate via TCP over a 2Gb Ethernet network. We found the communication latencies of TCP over Ethernet to be quite significant, and each of our nodes generally spent half of its time on communication. In practice, our algorithms would greatly benefit from a low-latency network (such as Infiniband).

We ran each of the three optimization algorithms with $k = 2, 5, 10$ worker nodes, and with a mini-batch size of $b = 5000$. We also experimented with different chunk sizes of $c = 500, 1000, 2500$, where c is constrained to be at most b/k. We evaluated each algorithm both as an online predictor (measuring average online loss) and as an online optimization algorithm (estimating test loss using the next mini-batch of examples, which is yet unobserved by the algorithm). We repeated each experiment 5 times, each time on a random permutation of the data, and we report averaged results in Fig. 1. The loss values are shown as a function of wall clock time (in log-scale).

The advantage of using more nodes is clearly reflected in our results. For example, take the ADMB algorithm in the online prediction setting. Fig. 1 clearly shows how more nodes accelerate the convergence of the average loss. Moreover, more nodes increase the throughput of the online prediction system, and the ADMB algorithm was able to process 248K, 140K and 76K examples per second, using 10, 5 and 2 workers respectively. Another example is the MaWo-DMB algorithm in the optimization setting. In order to achieve an average loss of 0.55, the MaWo-DMB algorithm required 34, 47 and 52 seconds, using 10, 5 and 2 workers respectively. Although we see a clear advantage to using more workers, the speedups do not scale linearly. With larger values of k, we begin to observe diminishing returns. The reason for this is the high communication latency associated with the TCP protocol over Ethernet.

We also conducted experiments with simulated node failures. In these experiments, each node flips an unbiased coin after processing each chunk of c examples. With probability half, the gradients from this chunk are shared with the rest of the cluster, as prescribed by the algorithm, and with probability half these gradients are discarded. In these experiments, we count only the loss of examples that were not discarded (this is also what we analyze theoretically). This reinforces our theoretical guarantees on the performance of our algorithms in the presence of node failures.

6. Conclusions

In this paper we presented the distributed mini-batches framework. Using this framework it is possible to achieve optimal regret bounds with smooth loss functions in the stochastic distributed setting. This result closes a gap in the theory of online learning and stochastic optimization. Since distributed environments often suffer from instabilities, we put an emphasis on methods that robustify the DMB framework.

References

Abernethy, J., Agarwal, A., Rakhlin, A., and Bartlett, P. A stochastic view of optimal regret through minimax duality. In *COLT*, 2009.

Bertsekas, D. and Tsitsiklis, J. *Parallel and Distributed Computation*. Prentice Hall, 1989.

Dekel, O., Gilad-Bachrach, R., Shamir, O., and Xiao, L. Robust distributed online prediction. arXiv, 2010a. URL http://arxiv.org/abs/1012.1370.

Dekel, O., Gilad-Bachrach, R., Shamir, O., and Xiao, L. Optimal distributed online prediction using mini-batches. arXiv, 2010b. URL http://arxiv.org/abs/1012.1367.

Delalleau, O. and Bengio, Y. Parallel stochastic gradient descent. Talk presented at CIFAR NCAP Summer School, Toronto, Canada, 2007.

Duchi, J., Agarwal, A., and Wainwright, M. Distributed dual averaging in networks. In *NIPS*, 2010.

Gallager, R., Humblet, P., and Spira, P. A distributed algorithm for minimum-weight spanning trees. *ACM TOPLAS Journal*, 1983.

Gimpel, K., Das, D., and Smith, N. Distributed asynchronous online learning for natural language processing. In *CoNLL*, 2010.

Langford, J., Smola, A., and Zinkevich, M. Slow learners are fast. In *NIPS*, 2009.

Malpani, N., Welch, J., and Waidya, N. Leader election algorithms for mobile ad hoc networks. In *DIAL-M*, 2000.

Nedić, A. and Ozdaglar, A. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

Nemirovski, A. and Yudin, D. *Problem complexity and method efficiency in optimization*. Series in Discrete Mathematics. Wiley-Interscience, 1983.

Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.

Nesterov, Y. Primal-dual subgradient methods for convex problems. *Mathematical Programming*, 120 (1):221–259, 2009.

Ram, S., Nedić, A., and Veeravalli, V. Distributed subgradient projection algorithm for convex optimization. In *ICASSP*, pp. 3653–3656, 2009.

Tsitsiklis, J., Bertsekas, D., and Athans, M. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812, 1986.

Xiao, L. Dual averaging methods for regularized stochastic learning and online optimization. *Journal of Machine Learning Research*, 11:2543–2596, 2010.

Zinkevich, M. Online convex programming and generalized infinitesimal gradient ascent. In *ICML*, 2003.

Zinkevich, M., Weimer, M., Smola, A., and Li, L. Parallelized stochastic gradient descent. In *NIPS*, 2010.