Slow Learners are Fast

John Langford, Alexander J. Smola, Martin Zinkevich
Machine Learning, Yahoo! Labs and Australian National University
4401 Great America Pky, Santa Clara, 95051 CA
{jl, maz, smola}@yahoo-inc.com

Abstract

Online learning algorithms have impressive convergence properties when it comes to risk minimization and convex games on very large problems. However, they are inherently sequential in their design which prevents them from taking advantage of modern multi-core architectures. In this paper we prove that online learning with delayed updates converges well, thereby facilitating parallel online learning.

1 Introduction

Online learning has become the paradigm of choice for tackling very large scale estimation problems. The convergence properties are well understood and have been analyzed in a number of different frameworks such as by means of asymptotics [12], game theory [8], or stochastic programming [13]. Moreover, learning-theory guarantees show that $O(1)$ passes over a dataset suffice to obtain optimal estimates [3, 2]. This suggests that online algorithms are an excellent tool for learning.

This view, however, is slightly deceptive for several reasons: current online algorithms process one instance at a time. That is, they receive the instance, make some prediction, incur a loss, and update an associated parameter. In other words, the algorithms are entirely sequential in their nature. While this is acceptable in single-core processors, it is highly undesirable given that the number of processing elements available to an algorithm is growing exponentially (e.g. modern desktop machines have up to 8 cores, graphics cards up to 1024 cores). It is therefore very wasteful if only one of these cores is actually used for estimation.

A second problem arises from the fact that network and disk I/O have not been able to keep up with the increase in processor speed. A typical network interface has a throughput of 100MB/s and disk arrays have comparable parameters. This means that current algorithms reach their limit at problems of size 1TB whenever the algorithm is I/O bound (this amounts to a training time of 3 hours), or even smaller problems whenever the model parametrization makes the algorithm CPU bound.

Finally, distributed and cloud computing are unsuitable for today's online learning algorithms. This creates a pressing need to design algorithms which break the sequential bottleneck. We propose two variants. To our knowledge, this is the first paper which provides *theoretical guarantees* combined with empirical evidence for such an algorithm. Previous work, e.g. by [6] proved rather inconclusive in terms of theoretical and empirical guarantees.

In a nutshell, we propose the following two variants: several processing cores perform stochastic gradient descent *independently* of each other while sharing a common parameter vector which is updated asynchronously. This allows us to accelerate computationally intensive problems whenever gradient computations are relatively expensive. A second variant assumes that we have linear function classes where parts of the function can be computed *independently* on several cores. Subsequently the results are combined and the combination is then used for a descent step.

A common feature of both algorithms is that the update occurs with some delay: in the first case other cores may have updated the parameter vector in the meantime, in the second case, other cores may have already computed parts of the function for the subsequent examples before an update.

2 Algorithm

2.1 Platforms

We begin with an overview of three platforms which are available for parallelization of algorithms. They differ in their structural parameters, such as synchronization ability, latency, and bandwidth and consequently they are better suited to different styles of algorithms. The description is not comprehensive by any means. For instance, there exist numerous variants of communication paradigms for distributed and cloud computing.

Shared Memory Architectures: The commercially available 4-16 core CPUs on servers and desktop computers fall into this category. They are general purpose processors which operate on a joint memory space where each of the processors can execute arbitrary pieces of code independently of other processors. Synchronization is easy via shared memory/interrupts/locks. A second example are graphics cards. There the number of processing elements is vastly higher (1024 on high-end consumer graphics cards), although they tend to be bundled into groups of 8 cores (also referred to as multiprocessing elements), each of which can execute a given piece of code in a data-parallel fashion. An issue is that explicit synchronization between multiprocessing elements is difficult — it requires computing kernels on the processing elements to complete.

Clusters: To increase I/O bandwidth one can combine several computers in a cluster using MPI or PVM as the underlying communications mechanism. A clear limit here is bandwidth constraints and latency for inter-computer communication. On Gigabit Ethernet the TCP/IP latency can be in the order of 100μs, the equivalent of 10^5 clock cycles on a processor and network bandwidth tends to be a factor 100 slower than memory bandwdith.

Grid Computing: Computational paradigms such as MapReduce [4] and Hadoop are well suited for the parallelization of batch-style algorithms [17]. In comparison to cluster configurations communication and latency are further constrained. For instance, often individual processing elements are unable to communicate directly with other elements with disk / network storage being the only mechanism of inter-process data transfer. Moreover, the latency is significantly increased.

We consider only the first two platforms since latency plays a critical role in the analysis of the class of algorithms we propose. While we do not exclude the possibility of devising parallel online algorithms suited to grid computing, we believe that the family of algorithm proposed in this paper is unsuitable and a significantly different synchronization paradigm would be needed.

2.2 Delayed Stochastic Gradient Descent

Many learning problems can be written as convex minimization problems. It is our goal to find some parameter vector x (which is drawn from some Banach space $\mathcal{X}$ with associated norm $\|\cdot\|$) such that the sum over convex functions $f_i : \mathcal{X} \to \mathbb{R}$ takes on the smallest value possible. For instance, (penalized) maximum likelihood estimation in exponential families with fully observed data falls into this category, so do Support Vector Machines and their structured variants. This also applies to distributed games with a communications constraint within a team.

At the outset we make no special assumptions on the order or form of the functions f_i. In particular, an adversary may choose to order or generate them in response to our previous choices of x. In other cases, the functions f_i may be drawn from some distribution (e.g. whenever we deal with induced losses). It is our goal to find a sequence of x_i such that the cumulative loss $\sum_i f_i(x_i)$ is minimized. With some abuse of notation we identify the average empirical and expected loss *both* by f^*. This is possible, simply by redefining $p(f)$ to be the uniform distribution over F. Denote by

$$f^*(x) := \frac{1}{|F|} \sum_i f_i(x) \text{ or } f^*(x) := \mathbf{E}_{f \sim p(f)}[f(x)] \tag{1}$$

$$\text{and correspondingly } x^* := \operatorname*{argmin}_{x \in \mathcal{X}} f^*(x) \tag{2}$$

the average risk. We assume that x^* exists (convexity does *not* guarantee a bounded minimizer) and that it satisfies $\|x^*\| \leq R$ (this is always achievable, simply by intersecting $\mathcal{X}$ with the unit-ball of radius R). We propose the following algorithm:

Algorithm 1 Delayed Stochastic Gradient Descent

Input: Feasible space $X \subseteq \mathbb{R}^n$, annealing schedule η_t and delay $\tau \in \mathbb{N}$
Initialization: set $x_1 \ldots, x_\tau = 0$ and compute corresponding $g_t = \nabla f_t(x_t)$.
for $t = \tau + 1$ **to** $T + \tau$ **do**
  Obtain f_t and incur loss $f_t(x_t)$
  Compute $g_t := \nabla f_t(x_t)$
  Update $x_{t+1} = \operatorname{argmin}_{x \in X} \|x - (x_t - \eta_t g_{t-\tau})\|$ (Gradient Step and Projection)
end for

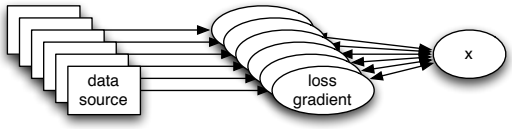

Figure 1: Data parallel stochastic gradient descent with shared parameter vector. Observations are partitioned on a per-instance basis among n processing units. Each of them computes its own gradient $g_t = \partial_x f_t(x_t)$. Since each computer is updating x in a round-robin fashion, it takes a delay of $\tau = n - 1$ between gradient computation and when the gradients are applied to x.

In this paper the annealing schedule will be either $\eta_t = \frac{\sigma}{(t-\tau)}$ or $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$. Often, $X = \mathbb{R}^n$. If we set $\tau = 0$, algorithm 1 becomes an entirely standard stochastic gradient descent algorithm. The only difference with delayed stochastic gradient descent is that we do *not* update the parameter vector x_t with the current gradient g_t but rather with a delayed gradient $g_{t-\tau}$ that we computed τ steps previously. We extend this to bounds which are dependent on strong convexity [1, 7] to obtain adaptive algorithms which can take advantage of well-behaved optimization problems in practice. An extension to Bregman divergences is possible. See [11] for details.

2.3 Templates

Asynchronous Optimization: Assume that we have n processors which can process data independently of each other, e.g. in a multicore platform, a graphics card, or a cluster of workstations. Moreover, assume that computing the gradient of $f_t(x)$ is at least n times as expensive as it is to update x (read, add, write). This occurs, for instance, in the case of conditional random fields [15, 18], in planning [14], and in ranking [19].

The rationale for delayed updates can be seen in the following setting: assume that we have n cores performing stochastic gradient descent on different instances f_t while sharing one common parameter vector x. If we allow each core in a round-robin fashion to update x one at a time then there will be a delay of $\tau = n - 1$ between when we see f_t and when we get to update $x_{t+\tau}$. The delay arises since updates by different cores cannot happen simultaneously. This setting is preferable whenever computation of f_t itself is time consuming.

Note that there is no need for explicit thread-level synchronization between individual cores. All we need is a read / write-locking mechanism for x or alternatively, atomic updates on the parameter vector. On a multi-computer cluster we can use a similar mechanism simply by having one server act as a state-keeper which retains an up-to-date copy of x while the loss-gradient computation clients can retrieve at any time a copy of x and send gradient update messages to the state keeper.

Pipelined Optimization: The key impediment in the previous template is that it required significant amounts of bandwidth solely for the purpose of synchronizing the state vector. This can be addressed by parallelizing computing the function value $f_i(x)$ explicitly rather than attempting to compute several instances of $f_i(x)$ simultaneously. Such situations occur, e.g. when $f_i(x) = g(\langle \phi(z_i), x \rangle)$ for high-dimensional $\phi(z_i)$. If we decompose the data z_i (or its features) over n nodes we can compute partial function values and also all partial updates *locally*. The only communication required is to combine partial values and to compute gradients with respect to $\langle \phi(z_i), x \rangle$.

This causes delay since the second stage is processing results of the first stage while the latter has already moved on to processing f_{t+1} or further. While the architecture is quite different, the effects are identical: the parameter vector x is updated with some delay τ. Note that here τ can be much smaller than the number of processors and mainly depends on the latency of the communication channel. Also note that in this configuration the memory access for x is entirely local.

Randomization: Order of observations matters for delayed updates: imagine that an adversary, aware of the delay τ bundles each of the τ most similar instances f_t together. In this case we will incur a loss that can be τ times as large as in the non-delayed case and require a learning rate which is τ times smaller. The reason being that only after seeing τ instances of f_t will we be able to respond to the data. Such highly correlated settings do occur in practice: for instance, e-mails or search keywords have significant temporal correlation (holidays, political events, time of day) and cannot be treated as iid data. Randomization of the order of data can be used to alleviate the problem.

3 Lipschitz Continuous Losses

Due to space constraints we only state the results and omit the proofs. A more detailed analysis can be found in [11]. We begin with a simple game theoretic analysis that only requires f_t to be *convex* and where the subdifferentials are bounded $\|\nabla f_t(x)\| \leq L$ by some $L > 0$. Denote by x^* the minimizer of $f^*(x)$. It is our goal to bound the regret R associated with a sequence $X = \{x_1, \ldots, x_T\}$ of parameters. If all terms are convex we obtain

$$R[X] := \sum_{t=1}^{T} f_t(x_t) - f_t(x^*) \leq \sum_{t=1}^{T} \langle \nabla f_t(x_t), x_t - x^* \rangle = \sum_{t=1}^{T} \langle g_t, x_t - x^* \rangle . \tag{3}$$

Next define a potential function measuring the distance between x_t and x^*. In the more general analysis this will become a Bregman divergence. We define $D(x\|x') := \frac{1}{2}\|x - x'\|^2$. At the heart of our regret bounds is the following which bounds the instantaneous risk at a given time [16]:

Lemma 1 *For all x^* and for all $t > \tau$, if $X = \mathbb{R}^n$, the following expansion holds:*

$$\langle x_{t-\tau} - x^*, g_{t-\tau} \rangle = \frac{1}{2}\eta_t \|g_{t-\tau}\|^2 + \frac{D(x^*\|x_t) - D(x^*\|x_{t+1})}{\eta_t} + \sum_{j=1}^{\min(\tau, t-(\tau+1))} \eta_{t-j} \langle g_{t-\tau-j}, g_{t-\tau} \rangle$$

Note that the decomposition of Lemma 1 is very similar to standard regret decomposition bounds, such as [21]. The key difference is that we now have an additional term characterizing the correlation between successive gradients which needs to be bounded. In the worst case all we can do is bound $\langle g_{t-\tau-j}, g_{t-\tau} \rangle \leq L^2$, whenever the gradients are highly correlated, which yields the following:

Theorem 2 *Suppose all the cost functions are Lipschitz continuous with a constant L and $\max_{x,x' \in X} D(x\|x') \leq F^2$. Given $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ for some constant $\sigma > 0$, the regret of the delayed update algorithm is bounded by*

$$R[X] \leq \sigma L^2 \sqrt{T} + F^2 \frac{\sqrt{T}}{\sigma} + L^2 \frac{\sigma \tau^2}{2} + 2L^2 \sigma \tau \sqrt{T} \tag{4}$$

and consequently for $\sigma^2 = \frac{F^2}{2\tau L^2}$ and $T \geq \tau^2$ we obtain the bound

$$R[X] \leq 4FL\sqrt{\tau T} \tag{5}$$

In other words the algorithm converges at rate $O(\sqrt{\tau T})$. This is similar to what we would expect in the worst case: an adversary may reorder instances such as to maximally slow down progress. In this case a parallel algorithm is no faster than a sequential code. This result may appear overly pessimistic but the following example shows that such worst-case scaling behavior is to be expected:

Lemma 3 *Assume that an optimal online algorithm with regard to a convex game achieves regret $R[m]$ after seeing m instances. Then any algorithm which may only use information that is at least τ instances old has a worst case regret bound of $\tau R[m/\tau]$.*

Our construction works by designing a sequence of functions f_i where for a fixed $n \in \mathbb{N}$ all $f_{n\tau+j}$ are identical (for $j \in \{1, \ldots, n\}$). That is, we send identical functions to the algorithm while it has no chance of responding to them. Hence, even an algorithm *knowing* that we will see τ identical instances in a row but being disallowed to respond to them for τ instances will do no better than one which sees every instance once but is allowed to respond instantly.

The useful consequence of Theorem 2 is that we are guaranteed to converge *at all* even if we encounter delay (the latter is not trivial — after all, we could end up with an oscillating parameter vector for overly aggressive learning rates). While such extreme cases hardly occur in practice, we need to make stronger assumptions in terms of correlation of f_t and the degree of smoothness in f_t to obtain tighter bounds. We conclude this section by studying a particularly convenient case: the setting when the functions f_i are strongly convex satisfying

$$f(x^*) \geq f_i(x) + \langle x^* - x, \partial_x f(x) \rangle + \frac{h_i}{2} \|x - x^*\|^2 \tag{6}$$

Here we can get rid of the $D(x^* \| x_1)$ dependency in the loss bound.

Theorem 4 *Suppose that the functions f_i are strongly convex with parameter $\lambda > 0$. Moreover, choose the learning rate $\eta_t = \frac{1}{\lambda(t-\tau)}$ for $t > \tau$ and $\eta_t = 0$ for $t \leq \tau$. Then under the assumptions of Theorem 2 we have the following bound:*

$$R[X] \leq \lambda \tau F^2 + \left[\tfrac{1}{2} + \tau\right] \frac{L^2}{\lambda} (1 + \tau + \log T) \tag{7}$$

The key difference is that now we need to take the additional contribution of the gradient correlations into account. As before, we pay a linear price in the delay τ.

4 Decorrelating Gradients

To improve our bounds beyond the most pessimistic case we need to assume that the adversary is not acting in the most hostile fashion possible. In the following we study the opposite case — namely that the adversary is drawing the functions f_i iid from an arbitrary (but fixed) distribution. The key reason for this requirement is that we need to control the value of $\langle g_t, g_{t'} \rangle$ for adjacent gradients.

The flavor of the bounds we use will be in terms of the *expected* regret rather than an actual regret. Conversions from expected to realized regret are standard. See e.g. [13, Lemma 2] for an example of this technique. For this purpose we need to take expectations of sums of copies of the bound of Lemma 1. Note that this is feasible since expectations are linear and whenever products between more than one term occur, they can be seen as products which are *conditionally independent* given past parameters, such as $\langle g_t, g_{t'} \rangle$ for $|t - t'| \leq \tau$ (in this case no information about g_t can be used to infer $g_{t'}$ or vice versa, given that we already know all the history up to time $\min(t, t') - 1$.

A key quantity in our analysis are bounds on the correlation between subsequent instances. In some cases we will only be able to obtain bounds on the *expected* regret rather than the actual regret. For the reasons pointed out in Lemma 3 this is an in-principle limitation of the setting.

Our first strategy is to assume that f_t arises from a scalar function of a linear function class. This leads to bounds which, while still bearing a linear penalty in τ, make do with considerably improved constants. The second strategy makes stringent smoothness assumptions on f_t, namely it assumes that the gradients themselves are Lipschitz continuous. This will lead to guarantees for which the delay becomes increasingly irrelevant as the algorithm progresses.

4.1 Covariance bounds for linear function classes

Many functions $f_t(x)$ depend on x only via an inner product. They can be expressed as

$$f_t(x) = l(y_t, \langle z_t, x \rangle) \text{ and hence } g_t(x) = \nabla f_t(x) = z_t \partial_{\langle z_t, x \rangle} l(y_t, \langle z_t, x \rangle) \tag{8}$$

Now assume that $\left|\partial_{\langle z_t, x \rangle} l(y_t, \langle z_t, x \rangle)\right| \leq \Lambda$ for all x and all t. This holds, e.g. in the case of logistic regression, the soft-margin hinge loss, novelty detection. In all three cases we have $\Lambda = 1$. Robust loss functions such as Huber's regression score [9] also satisfy (8), although with a different constant (the latter depends on the level of robustness). For such problems it is possible to bound the correlation between subsequent gradients via the following lemma:

Lemma 5 *Denote by $(y, z), (y', z') \sim \Pr(y, z)$ random variables which are drawn independently of $x, x' \in \mathcal{X}$. In this case*

$$\mathbf{E}_{y,z,y',z'} \left[\langle \partial_x l(y, \langle z, x \rangle), \partial_x l(y', \langle z', x' \rangle) \rangle\right] \leq \Lambda^2 \left\| \mathbf{E}_{z,z'} \left[z' z^\top\right] \right\|_{\text{Frob}} =: L^2 \alpha \tag{9}$$

Here we defined α to be the scaling factor which quantifies by how much gradients are correlated. This yields a tighter version of Theorem 2.

Corollary 6 *Given $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ and the conditions of Lemma 5 the regret of the delayed update algorithm is bounded by*

$$R[X] \leq \sigma L^2 \sqrt{T} + F^2 \frac{\sqrt{T}}{\sigma} + L^2 \alpha \frac{\sigma \tau^2}{2} + 2L^2 \alpha \sigma \tau \sqrt{T} \tag{10}$$

Hence for $\sigma^2 = \frac{F^2}{2\tau\alpha L^2}$ (assuming that $\tau\alpha \geq 1$) and $T \geq \tau^2$ we obtain $R[X] \leq 4FL\sqrt{\alpha\tau T}$.

4.2 Bounds for smooth gradients

The key to improving the *rate* rather than the *constant* with regard to which the bounds depend on τ is to impose further smoothness constraints on f_t. The rationale is quite simple: we want to ensure that small changes in x do not lead to large changes in the gradient. This is precisely what we need in order to show that a small delay (which amounts to small changes in x) will not impact the update that is carried out to a significant amount. More specifically we assume that the gradient of f is a Lipschitz-continuous function. That is,

$$\|\nabla f_t(x) - \nabla f_t(x')\| \leq H \|x - x'\| . \tag{11}$$

Such a constraint effectively rules out piecewise linear loss functions, such as the hinge loss, structured estimation, or the novelty detection loss. Nonetheless, since this discontinuity only occurs on a set of measure 0 delayed stochastic gradient descent still works very well on them in practice.

Theorem 7 *In addition to the conditions of Theorem 2 assume that the functions f_i are i.i.d., $H \geq \frac{L}{4F\sqrt{\tau}}$ and that H also upper-bounds the change in the gradients as in Equation 11. Moreover, assume that we choose a learning rate $\eta_t = \frac{\sigma}{\sqrt{t-\tau}}$ with $\sigma = \frac{F}{L}$. In this case the risk is bounded by*

$$\mathbf{E}[R[X]] \leq \left[28.3F^2H + \frac{2}{3}FL + \frac{4}{3}F^2H \log T\right] \tau^2 + \frac{8}{3}FL\sqrt{T}. \tag{12}$$

Note that the convergence bound which is $O(\tau^2 \log T + \sqrt{T})$ is governed by two different regimes. Initially, a delay of τ can be quite harmful since subsequent gradients are highly correlated. At a later stage when optimization becomes increasingly an *averaging* process a delay of τ in the updates proves to be essentially harmless. The key difference to bounds of Theorem 2 is that now the *rate* of convergence has improved dramatically and is essentially as good as in sequential online learning. Note that H does not influence the *asymptotic* convergence properties but it significantly affects the initial convergence properties.

This is exactly what one would expect: initially while we are far away from the solution x^* parallelism does not help much in providing us with guidance to move towards x^*. However, after a number of steps online learning effectively becomes an averaging process for variance reduction around x^* since the stepsize is sufficiently small. In this case averaging becomes the dominant force, hence parallelization does not degrade convergence further. Such a setting is desirable — after all, we want to have good convergence for extremely large amounts of data.

4.3 Bounds for smooth gradients with strong convexity

We conclude this section with the tightest of all bounds — the setting where the losses are all strongly convex and smooth. This occurs, for instance, for logistic regression with ℓ_2 regularization. Such a requirement implies that the objective function $f^*(x)$ is sandwiched between two quadratic functions, hence it is not too surprising that we should be able to obtain rates comparable with what is possible in the minimization of quadratic functions. Also note that the ratio between upper and lower quadratic bound loosely corresponds to the condition number of a quadratic function — the ratio between the largest and smallest eigenvalue of the matrix involved in the optimization problem.

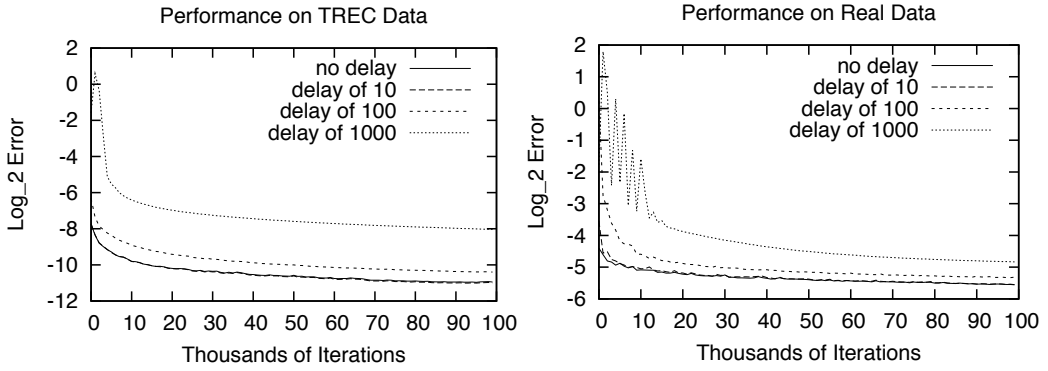

Figure 2: Experiments with simulated delay on the TREC dataset (left) and on a propietary dataset (right). In both cases a delay of 10 has no effect on the convergence whatsoever and even a delay of 100 is still quite acceptable.

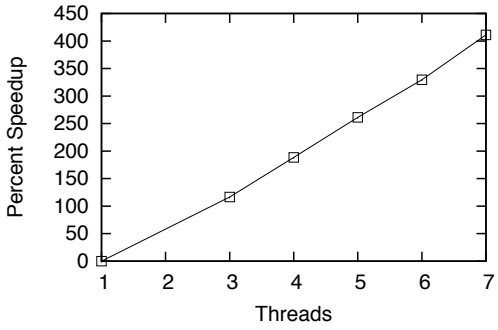

Figure 3: Time performance on a subset of the TREC dataset which fits into memory, using the quadratic representation. There was either one thread (a serial implementation) or 3 or more threads (master and 2 or more slaves).

Theorem 8 *Under the assumptions of Theorem 4, in particular, assuming that all functions f_i are i.i.d and strongly convex with constant λ and corresponding learning rate $\eta_t = \frac{1}{\lambda(t-\tau)}$ and provided that Equation 11 holds we have the following bound on the expected regret:*

$$\mathbf{E}\left[R[X]\right] \leq \frac{10}{9}\left[\lambda\tau F^2 + \left[\frac{1}{2}+\tau\right]\frac{L^2}{\lambda}[1+\tau+\log(3\tau+(H\tau/\lambda))] + \frac{L^2}{2\lambda}[1+\log T] + \frac{\pi^2\tau^2 HL^2}{6\lambda^2}\right]. \tag{13}$$

As before, this improves the rate of the bound. Instead of a dependency of the form $O(\tau \log T)$ we now have the dependency $O(\tau \log \tau + \log T)$. This is particularly desirable for large T. We are now within a small factor of what a fully sequential algorithm can achieve. In fact, we could make the constant arbitrary small for large enough T.

5 Experiments

In our experiments we focused on pipelined optimization. In particular, we used two different training sets that were based on e-mails: the TREC dataset [5], consisting of 75,419 e-mail messages, and a proprietary (significantly harder) dataset of which we took 100,000 e-mails. These e-mails were tokenized by whitespace. The problem there is one of binary classification where we minimized a 'Huberized' soft-margin loss function

$$f_t(x) = l(y_t \langle z_t, x\rangle) \text{ where } l(\chi) = \begin{cases} \frac{1}{2} - \chi & \text{if } \chi \leq 0 \\ \frac{1}{2}(\chi-1)^2 & \text{if } \chi \in [0,1] \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

Here $y_t \in \{\pm 1\}$ denote the labels of the binary classification problem, and l is the smoothed quadratic soft-margin loss of [10]. We used two feature representations: a linear one which amounted to a simple bag of words representation, and a quadratic one which amounted to generating a bag of word pairs (consecutive or not).

To deal with high-dimensional feature spaces we used hashing [20]. In particular, for the TREC dataset we used 2^{18} feature bins and for the proprietary dataset we used 2^{24} bins. Note that hashing comes with performance guarantees which state that the canonical distortion due to hashing is sufficiently small for the dimensionality we picked. We tried to address the following issues:

1. The obvious question is a systematic one: how much of a convergence penalty do we incur in practice due to delay. This experiment checks the goodness of our bounds. We checked convergence for a system where the delay is given by $\tau \in \{0, 10, 100, 1000\}$.
2. Secondly, we checked on an actual parallel implementation whether the algorithm scales well. Unlike the previous check includes issues such as memory contention, thread synchronization, and general feasibility of a delayed updating architecture.

Implementation The code was written in Java, although several of the fundamentals were based upon VW [10], that is, hashing and the choice of loss function. We added regularization using lazy updates of the parameter vector (i.e. we rescale the updates and occasionally rescale the parameter). This is akin to Leon Bottou's SGD code. For robustness, we used $\eta_t = \frac{1}{\sqrt{t}}$.

All timed experiments were run on a single, 8 core machine with 32 GB of memory. In general, at least 6 of the cores were free at any given time. In order to achieve advantages of parallelization, we divide the feature space $\{1 \dots n\}$ into roughly equal pieces, and assign a slave thread to each piece. Each slave is given both the weights for its pieces, as well as the corresponding pieces of the examples. The master is given the label of each example. We compute the dot product separately on each piece, and then send these results to a master. The master adds the pieces together, calculates the update, and then sends that back to the slaves. Then, the slaves update their weight vectors in proportion to the magnitude of the central classifier. What makes this work quickly is that there are multiple examples in flight through this dataflow simultaneously. Note that between the time when a dot product is calculated for an example and when the results have been transcribed, the weight vector has been updated with several other earlier examples and the dot products have been calculated from several later examples. As a safeguard we limited the maximum delay to 100 examples. In this case the compute slave would simply wait for the pipeline to clear.

The first experiment that we ran was a simulation where we artificially added a delay between the update and the product (Figure 2a). We ran this experiment using linear features, and observed that the performance did not noticeably degrade with a delay of 10 examples, did not significantly degrade with a delay of 100, but with a delay of 1000, the performance became much worse.

The second experiment that we ran was with a proprietary dataset (Figure 2b). In this case, the delays hurt less; we conjecture that this was because the information gained from each example was smaller. In fact, even a delay of 1000 does not result in particularly bad performance.

Since even the sequential version already handled 150,000 examples per second we tested parallelization only for quadratic features where throughput would be in the order of 1000 examples per second. Here parallelization dramatically improved performance — see Figure 3. To control for disk access we loaded a subset of the data into memory and carried out the algorithm on it.

Summary and Discussion

The type of updates we presented is a rather natural one. However, intuitively, having a delay of τ is like having a learning rate that is τ times larger. In this paper, we have shown theoretically how independence between examples can make the actual effect much smaller.

The experimental results showed three important aspects: first of all, small simulated delayed updates do not hurt much, and in harder problems they hurt less; secondly, in practice it is hard to speed up "easy" problems with a small amount of computation, such as e-mails with linear features; finally, when examples are larger or harder, the speedups can be quite dramatic.

References

[1] Peter L. Bartlett, Elad Hazan, and Alexander Rakhlin. Adaptive online gradient descent. In J. C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, Cambridge, MA, 2008. MIT Press.

[2] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In J. C. Platt, D. Koller, Y. Singer, and S.T. Roweis, editors, *NIPS*. MIT Press, 2007.

[3] Léon Bottou and Yann LeCun. Large scale online learning. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 217–224, Cambridge, MA, 2004. MIT Press.

[4] C.T. Chu, S.K. Kim, Y. A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. Platt, and T. Hofmann, editors, *Advances in Neural Information Processing Systems 19*, 2007.

[5] G. Cormack. TREC 2007 spam track overview. In *The Sixteenth Text REtrieval Conference (TREC 2007) Proceedings*, 2007.

[6] O. Delalleau and Y. Bengio. Parallel stochastic gradient descent, 2007. CIAR Summer School, Toronto.

[7] C.B. Do, Q.V. Le, and C.-S. Foo. Proximal regularization for online and batch learning. In A.P. Danyluk, L. Bottou, and M. L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382, page 33. ACM, 2009.

[8] Elad Hazan, Amit Agarwal, and Satyen Kale. Logarithmic regret algorithms for online convex optimization. *Machine Learning*, 69(2-3):169–192, 2007.

[9] P. J. Huber. *Robust Statistics*. John Wiley and Sons, New York, 1981.

[10] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007. `http://hunch.net/?p=309`.

[11] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. arXiv:0911.0491.

[12] N. Murata, S. Yoshizawa, and S. Amari. Network information criterion — determining the number of hidden units for artificial neural network models. *IEEE Transactions on Neural Networks*, 5:865–872, 1994.

[13] Y. Nesterov and J.-P. Vial. Confidence level solutions for stochastic programming. Technical Report 2000/13, Université Catholique de Louvain - Center for Operations Research and Economics, 2000.

[14] N. Ratliff, J. Bagnell, and M. Zinkevich. Maximum margin planning. In *International Conference on Machine Learning*, July 2006.

[15] N. Ratliff, J. Bagnell, and M. Zinkevich. (online) subgradient methods for structured prediction. In *Eleventh International Conference on Artificial Intelligence and Statistics (AIStats)*, March 2007.

[16] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated subgradient solver for SVM. In *Proc. Intl. Conf. Machine Learning*, 2007.

[17] Choon Hui Teo, S. V. N. Vishwanthan, Alex J. Smola, and Quoc V. Le. Bundle methods for regularized risk minimization. *J. Mach. Learn. Res.*, 2009. Submitted in February 2009.

[18] S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark Schmidt, and Kevin Murphy. Accelerated training conditional random fields with stochastic gradient methods. In *Proc. Intl. Conf. Machine Learning*, pages 969–976, New York, NY, USA, 2006. ACM Press.

[19] M. Weimer, A. Karatzoglou, Q. Le, and A. Smola. Cofi rank - maximum margin matrix factorization for collaborative ranking. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008.

[20] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A.J. Smola. Feature hashing for large scale multitask learning. In L. Bottou and M. Littman, editors, *International Conference on Machine Learning*, 2009.

[21] M. Zinkevich. Online convex programming and generalised infinitesimal gradient ascent. In *Proc. Intl. Conf. Machine Learning*, pages 928–936, 2003.