A zealous parallel gradient descent algorithm

Gilles Louppe and Pierre Geurts
Department of EE and CS & GIGA
University of Liège
Sart Tilman B28, B-4000 Liège, Belgium
{g.louppe, p.geurts}@ulg.ac.be

Abstract

Parallel and distributed algorithms have become a necessity in modern machine learning tasks. In this work, we focus on parallel asynchronous gradient descent [1, 2, 3] and propose a zealous variant that minimizes the idle time of processors to achieve a substantial speedup. We then experimentally study this algorithm in the context of training a restricted Boltzmann machine on a large collaborative filtering task.

1 Introduction

Modern advances and cost decreases in storage capacity, communication networks and instrumentations have led to the generation of massive datasets in domains as varied as the world-wide web, finance or life sciences. As a direct consequence, the size of typical machine learning tasks has greatly increased and traditional machine learning techniques are no longer suited to handle them properly. First, these datasets are so huge that they simply can no longer fit into the memory of classic single computers. Second, even if they could, most of conventional algorithms would struggle to handle these large problems within reasonnable time. To tackle these major issues, parallel and distributed machine learning algorithms constitute an inevitable direction of research.

In practice, online and mini-batch gradient descent algorithms already solve the memory issue. By design, they can indeed already tackle any large dataset since they don't require to load at once all the training samples into memory. However, those algorithms are inherently serial and hence are still prone to prohibitive computing times. In this work, we try to overcome this problem by parallelizing the computation across several processors in the context of shared memory architectures.

2 Mini-batch gradient descent

In many machine learning algorithms, the task of training a model often reduces to an optimization problem whose objective is to minimize $\mathbb{E}_z[C(\theta, z)]$ where C is some (typically convex) cost function and the expectation is computed over training points z. In batch gradient descent, the vector of parameters θ is iteratively updated after each pass over the whole set of training examples, using

$$\theta_{k+1} := \theta_k - \alpha \sum_{t=1}^{n} \frac{\partial C(\theta_k, z_t)}{\partial \theta}, \tag{1}$$

where α is some learning rate. When the training set is large though, each iteration may become very expensive and convergence may be slow. By contrast, in online (or *stochastic*) gradient descent, the algorithm sweeps through the training set, updating θ at each training example using

$$\theta_{t+1} := \theta_t - \alpha \frac{\partial C(\theta_t, z_t)}{\partial \theta}. \tag{2}$$

This usually leads to faster convergence, but also to noisy updates of the parameters, which may impair optimality in the long run. A compromise between the two approaches is to update the vector of parameters after a small number b of training examples (called a *mini-batch*) using

$$\theta_{k+1} := \theta_k - \alpha \sum_{t=s_k}^{s_k+b} \frac{\partial C(\theta_k, z_t)}{\partial \theta}. \tag{3}$$

In that way, gradients are less noisy (since they are averaged over b training examples) and convergence remains fast. Note that in practice, Eq. 1, 2 and 3 can also be enhanced with a regularization term.

3 Zealous parallel gradient descent

In a shared-memory environment, mini-batch gradient descent can be parallelized as proposed in [1, 2, 3]. In this setting, θ is stored in shared memory and multiple mini-batches are processed asynchronously and independently by multiple processors. Once a processor finishes its current mini-batch, it updates θ in mutual exclusion using a synchronization lock and then proceeds to the next mini-batch until some convergence criterion is met. As noted by [4] and [2], this algorithm do not simulate the same mini-batch procedure that would happen on a single processor. In particular, some delay might occur between the time gradient components are computed and the time they are eventually used to update θ. Put otherwise, this amounts to say that processors might use slightly stale parameters that do not take into account the very last updates. Yet, [1] showed that convergence is still guaranteed under some conditions, and [4] derived convergence rate results in the case of stochastic gradient descent.

The asynchronous algorithm presented above works fairly well in practice but suffers from a bottleneck which impairs the gains due to the parallelization. In not so uncommon cases, contention might indeed appear on the synchronization lock, hence causing the processors to queue and idle. For example, when the number of parameters is very large, updating θ do no longer take a negligeable time and contention is likely to happen. Likewise, the more processors, the more likely they are to queue on the synchronization lock.

To solve this problem, we propose the following more resource-efficient scheme. First, instead of blocking on the synchronization lock if some other processor is already in the critical section, we make processors skip the update altogether. In that case, a processor directly and zealously proceeds to the next mini-batch and locally queues its updates until it eventually enters the critical section. Second, the access policy to the critical section is changed so that access is granted only to the processor with the most queued updates. The combination of these two strategies prevents processors from idling and limits at the same time the effects of increasing the delay between updates of θ. The procedure is summarized below in algorithm 1.

```
shared $\theta$;
$\Delta\theta \leftarrow 0$;
while not converged($\theta$) do
    mini-batch ← get next mini-batch;
    for all z ∈ mini-batch do
        $\Delta\theta \leftarrow \Delta\theta + \frac{\partial C(\theta,z)}{\partial\theta}$;
    end for
    if trylock(pid) then
        $\theta \leftarrow \theta - \alpha\Delta\theta$;
        $\Delta\theta \leftarrow 0$;
        next(pid);
    end if
end while
```

```
shared next ← 0;
shared counter ← new int[np];
function trylock(pid)
    counter[pid] ← counter[pid] + 1;
    return next == pid;
end function
function next(pid)
    counter[pid] ← 0;
    next ← arg max(counter);
end function
```

Algorithm 1 Zealous parallel gradient descent. The left snippet of code defines the procedure followed by each individual processor (or thread). The right snippet of code defines the functions used to implement the policy granting access to the critical section to the processor with the most queued updates. In this pseudo-code, np is the number of threads and pid is an identifier unique to each thread (ranging from 0 to $np - 1$).

4 Results

The experiments presented below measure the speedup obtainable by our zealous parallel gradient descent algorithm in comparison with the speedup achieved by the asynchronous algorithm of [1, 2, 3]. The effects of delaying the updates of θ on the convergence of the method were also observed and evaluated.

We chose to evaluate our algorithm in the context of training a (conditional) restricted Boltzmann machine on a large collaborative filtering task, as introduced in [3, 5]. Training was performed on the Netflix dataset, which is known to be one of the largest datasets in the domain, using 100 hidden units and mini-batches of 500 users. Weights, biases of the visible units, biases of the hidden units and the D_{ij} elements were respectively updated using a learning rate of 0.0015, 0.0012, 0.1 and 0.001. Weight decay was set to 0.0001. No momentum heuristic was used. In that typical setting, θ counts 10750950 parameters, which makes updates actually fairly expensive in terms of computing time and hence increases the likelihood of blocking on the synchronization lock. All of our experiments were carried out on a dedicated 64-bit Linux machine with 24 1.9Ghz-cores and a total of 24GB of RAM.

Figure 1 illustrates the speedup and the parallel efficiency of both asynchronous and zealous algorithms. The speedup is the ratio between the execution time required for the sequential algorithm to perform a single pass over the whole learning set, and the execution time required for its parallel counterpart. The goal of this metric is to measure how much the parallel algorithms are faster than the sequential algorithm. The parallel efficiency is the speedup divided by the number of processors. It measures how well the parallel algorithms benefit from extra processors. We first observe that the zealous algorithm does indeed execute faster than the asynchronous algorithm. With 4 cores, the parallel efficiency of the zealous algorithm is nearly optimal, and then remains as expected significantly higher than the efficiency of the asynchronous algorithm as the number of cores increases. For both algorithms, we also find that the gains in terms of speedup decrease as the number of cores increases. This is actually not surprising and can be explained using Amdahl's argument [6]: the speedup of a parallel algorithm is bounded by the portion of code which is inherently serial. In our case, updates of θ remain serial, which explains the limit observed.

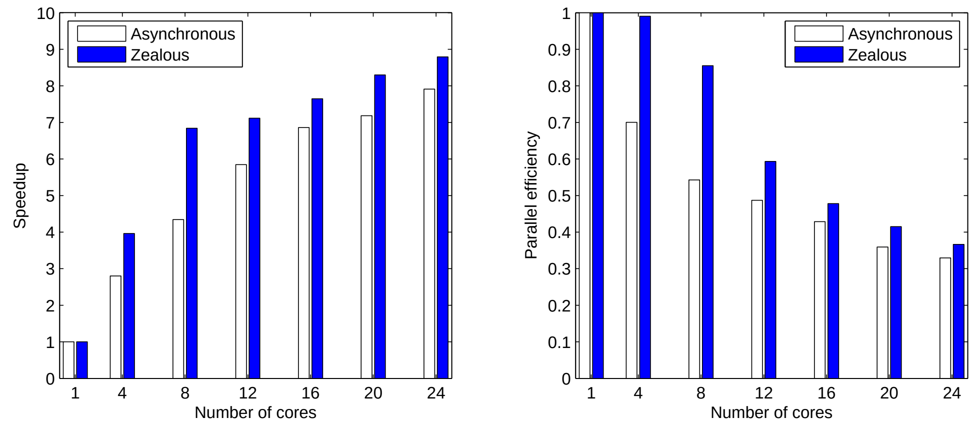

Figure 1: Speedup and parallel efficiency of the asynchronous and zealous algorithms.

Figure 2 illustrates the error curve of the model on an independent dataset (i.e., the probe set of the Netflix dataset) when trained with an increasing number of cores. It highlights the fact that the zealous algorithm does indeed converge faster than the asynchronous algorithm in terms of wall clock time. For instance, we observe that the zealous algorithm run with 4 and 8 cores converges nearly as fast as the asynchronous algorithm run with 8 and 16 cores. For 20 and 24 cores however, significant oscillations can be observed in the learning curves of the zealous algorithm. Actually, this is a manifestation of the effects of delaying the updates of θ. Indeed, the more cores there are, the more likely they will skip the update of θ and hence the more they will accumulate gradient components. This accumulation of stale gradients causes inertia towards old directions and becomes more and more harmful as the algorithm approaches to the optimum. This explains the oscillations and also why they only appear after some time. At a smaller scale, this phenomenon actually also occurs when using less cores or even in the asynchronous algorithm. The difference is that in those cases, processors do not accumulate enough gradients for the effects of the delay to become noticeable and to impair convergence.

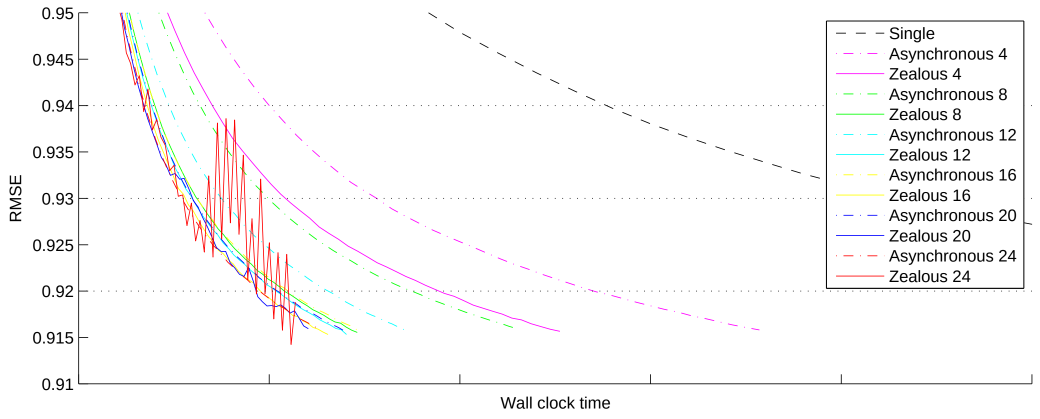

Figure 2: Learning curves of the asynchronous and zealous algorithms.

5 Conclusions

In this work, we have proposed a zealous parallel gradient descent algorithm that has shown significant speedup over the asynchronous parallel gradient algorithm of [1, 2, 3]. In particular, our algorithm shows appealing properties when θ is large and do not take a negligeable time to update. We have however also observed that due to the effects of delaying the updates of θ, our algorithm can impair convergence if the number of cores become too large. In future work, we would like to explore strategies to counter the effect of the delay and to redo more thorough experiments, on several other machine learning tasks, in order to corroborate the results obtained in this work. We also plan to investigate other distributed architectures in continuation to our work in the context of collaborative filtering [3].

Acknowledgments

Gilles Louppe and Pierre Geurts are respectively research fellow and research associate of the FNRS, Belgium. This paper presents research results of the Belgian Network BIOMAGNET (Bioinformatics and Modeling: from Genomes to Networks), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office. The scientific responsibility rests with its authors.

References

[1] A. Nedic, D.P. Bertsekas, and V.S. Borkar. Distributed asynchronous incremental subgradient methods. *Studies in Computational Mathematics*, 8:381–407, 2001.

[2] K. Gimpel, D. Das, and N.A. Smith. Distributed asynchronous online learning for natural language processing. In *Proceedings of the Conference on Computational Natural Language Learning*, 2010.

[3] G. Louppe. Collaborative filtering: Scalable approaches using restricted Boltzmann machines. Master's thesis, University of Liège, 2010.

[4] Martin Zinkevich, Alex Smola, and John Langford. Slow learners are fast. In *Advances in Neural Information Processing Systems 22*, pages 2331–2339. 2009.

[5] R. Salakhutdinov, A. Mnih, and G. E. Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, page 798. ACM, 2007.

[6] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.