异步方法在深度学习中得到了广泛的应用，但在非凸问题上的理论证明却很有限。

Asynchrony begets Momentum, with an Application to Deep Learning

Ioannis Mitliagkas
Dept. of Computer Science
Stanford University
Email: imit@stanford.edu

Ce Zhang
Dept. of Computer Science
ETH, Zurich
Email: ce.zhang@inf.ethz.ch

Stefan Hadjis, Christopher Ré
Dept. of Computer Science
Stanford University
Email: {shadjis, chrismre}@stanford.edu

***Abstract*—Asynchronous methods are widely used in deep learning, but have limited theoretical justification when applied to non-convex problems. We show that running stochastic gradient descent (SGD) in an asynchronous manner can be viewed as adding a momentum-like term to the SGD iteration. Our result does not assume convexity of the objective function, so it is applicable to deep learning systems. We observe that a standard queuing model of asynchrony results in a form of momentum that is commonly used by deep learning practitioners. This forges a link between queuing theory and asynchrony in deep learning systems, which could be useful for systems builders. For convolutional neural networks, we experimentally validate that the degree of asynchrony directly correlates with the momentum, confirming our main result. An important implication is that tuning the momentum parameter is important when considering different levels of asynchrony. We assert that properly tuned momentum reduces the number of steps required for convergence. Finally, our theory suggests new ways of counteracting the adverse effects of asynchrony: a simple mechanism like using negative algorithmic momentum can improve performance under high asynchrony. Since asynchronous methods have better hardware efficiency, this result may shed light on when asynchronous execution is more efficient for deep learning systems.**

I. Introduction

Stochastic Gradient Descent (SGD) and its variants are the optimization method of choice for many large-scale learning problems including deep learning [1, 2, 3, 4]. A popular approach to running these systems removes locks and synchronization barriers [5]. Such methods are called *asynchronous-parallel methods* or Hogwild! and are used on many systems by large companies like Microsoft and Google [6, 7].

However, the effectiveness of asynchrony is a bit of a mystery. For convex problems on sparse data, these race conditions do not slow down convergence too much [5], and the lack of locking means that each step takes less time. However, sparsity could not be the complete story as many groups have reported that asynchronous-parallel can be faster even for dense data [7, 6], in which case available theory [5, 8] does not apply. Recent work includes results in the dense case, for general convex problems asymptotically [9], and specifically for matrix completion problems [10].

In deep learning there has been a debate about how to scale up training. Many systems have run asynchronously [6, 7], but some have proposed that synchronous training may be faster [11, 12, 13]. As part of our work [14], we realized that often many systems do not tune the *momentum* parameter [15], and this can change the results drastically. Among deep learning practitioners—and some theoreticians—"momentum" is a synonym for 0.9. This is evidenced by the large number of papers and tutorials that prescribe it [16, 13, 17, 18] and by the fact that many high-quality publications [12, 11] do not report the value used, supporting the understanding that 0.9 has almost reached "industry standard" status. That said, there are some papers reporting results after tuning momentum, e.g., [7].

Like the step size, the best value for the momentum parameter depends on the objective, the data, and the underlying hardware. Until now, there was not much reason to think that the optimization and system dynamics interact, although this is folklore among mathematical optimization researchers. In this paper we provide theoretical and experimental evidence that the parameters and asynchrony interact in a precise way. We summarize our contributions:

- We show that asynchrony introduces momentum to the SGD update, called the *implicit momentum.*
- We argue that tuning the algorithmic momentum parameter [15], is important when considering different scales of asynchrony.
- Under a simple model, we describe exactly how implicit and *explicit* (algorithmic) momentum interact when both are present. We see numerically that under heavy asynchrony, *negative values of explicit momentum* are actually optimal.
- We verify all of these results with experiments on convolutional neural networks on our prototype system [14].

II. Preliminaries

Our aim is to minimize an objective $f : \mathbb{R}^d \to \mathbb{R}$, of the form

$$f(w) \triangleq \frac{1}{n}\sum_{i=1}^{n} f_i(w) = \frac{1}{n}\sum_{i=1}^{n} f(w; z_i). \tag{1}$$

Typically the component function $f_i(w)$ represents a loss function evaluated on a specific data point or mini-batch z_i. SGD considers one term at a time, calculates its gradient and uses it to update vector w.

$$w_{t+1} = w_t - \alpha_t \nabla_w f(w_t; z_{i_t}) \tag{2}$$

Sequence $(i_t)_t$ describes the order in which the samples are considered and $(\alpha_t)_t$ are the step sizes used.

1) Momentum: Introduced by Polyak [15], the momentum algorithm is ubiquitous in deep learning implementations, as it is known to offer a significant optimization boost [19]. For some $\mu_L \in [0, 1)$ it takes the following form.

$$w_{t+1} - w_t = \mu_L(w_t - w_{t-1}) - \alpha_t \nabla_w f(w_t; z_{i_t}) \tag{3}$$

In this paper we call μ_L the *explicit momentum*; the reason for this name to become clear soon.

2) Asynchrony: A popular way of parallelizing SGD is the fully asynchronous execution of (2) by M different workers, also known as HOGWILD! [5]. In the simplest topology, all workers share access to a main parameter store; either in main memory or a parameter server. Worker processes operate on potentially stale values of w. Let v_t denote the value read by the worker in charge of update t. Then

$$v_t = w_{t-\tau_t}, \tag{4}$$

for some random delay, τ_t, called *staleness*. This is referred to as the *consistent reads* model [8] and it makes sense for the dense updates and batch size typically used in CNNs. For the ensuing analysis, we make the assumption that, for every worker and sample t, the read delays are independent and follow a distribution denoted by Q. Specifically,

$$v_t = w_{t-l} \qquad \text{w.p.} \quad q_l, \quad l \in \mathbb{N}, \tag{5}$$

and the update step becomes

$$w_{t+1} = w_t - \alpha_t \nabla_w f(v_t; z_{i_t}), \tag{6}$$

where the gradient is taken with respect to the first argument of f.

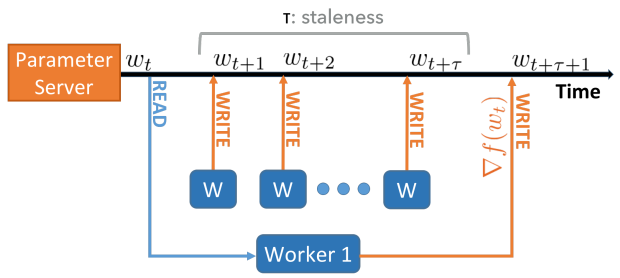

Fig. 1. Staleness is the number of writes in between a worker's read and write operation. We model it as a random variable τ.

III. Asynchrony begets momentum

The stochastic noise of SGD has been shown to have a stabilizing effect [20]. The effect of asynchrony, on the other hand, is not well understood and is often assumed to act as a regularizer. We show that it actually acts as an extra momentum term. We call this the *asynchrony-induced or implicit momentum* to differentiate from the explicit momentum introduced in Section II. Our experimental findings in Section IV support this intuition. Specifically, we see that the optimal value for the explicit momentum drops as we increase the number of workers. Our understanding is that in those cases, asynchrony contributes the missing momentum. We also see that tuning the momentum decreases the number of steps required to reach a target loss.

According to the model described in (5), the value v_t—read and used for the evaluation of step t—is a random variable sampled from the model's history, $(w_s)_{s \leq t}$. For example, the expectation of the value read is a convex combination of past values, $\mathbb{E}[v_t] = \sum_{l=0}^{\infty} q_l \mathbb{E}[w_{t-l}]$. This implies the existence of memory in an asynchronous system. In the following theorem we make this intuition rigorous. Some proofs are included in Appendix A; the rest can be found in the full version of this paper [21].

Assumption 1 (Staleness and example selection are independent)**.** *We make the following assumption on staleness.*

(A1) *The staleness process, $(\tau_t)_t$, and the sample selection process, $(i_t)_t$, are mutually independent.*

This assumption is valid on a CNN that performs dense updates, where the randomness in staleness comes from unmodeled implementation and system behavior.

Theorem 2 (Memory from asynchrony)**.** *Under Assumption 1 and for a constant step size $\alpha_t = \alpha$, we get the following momentum-like expression for consecutive* *updates.*

$$\begin{aligned}\mathbb{E}[w_{t+1} - w_t] =& \mathbb{E}[w_t - w_{t-1}] - \alpha q_0 \mathbb{E}\nabla f(w_t) \\ & + \alpha \sum_{l=0}^{\infty} (q_l - q_{l+1}) \mathbb{E}\nabla f(w_{t-(l+1)})\end{aligned}$$

This theorem suggests that, when staleness has strictly positive variance, the previous step contributes positively to the current step. Next we show that when staleness is geometrically distributed, we get the familiar form of momentum, discussed in Section II.

Theorem 3 (Momentum from geometric staleness)**.** *Let the staleness distribution be geometric on* $\{0, 1, \ldots\}$ *with parameter* $1 - \mu_S$*, i.e.* $q_l = (1 - \mu_S)\mu_S^l$*. The expected update takes the momentum form of* (3).

$$\begin{aligned}\mathbb{E}[w_{t+1} - w_t] =& \mu_S \mathbb{E}[w_t - w_{t-1}] \\ & - (1 - \mu_S)\alpha \mathbb{E}\nabla_w f(w_t) \qquad (7)\end{aligned}$$

A. Queuing Model

Here we show that under a simple queuing model, the conditions of Theorem 3 are satisfied. We denote the time it takes step t to finish W_t, and call $(W_t)_t$ the *work process*.

Assumption 4 (Independent, Exponential Work)**.** *We make the following assumptions on the work process.*

(A2) $W_t \sim \text{Exp}(\lambda)$

(A3) $(W_t)_t$ *are mutually independent.*

Theorem 5. *Consider* M *asynchronous workers and let* $(W_t)_t$ *denote the work process. If the* W_t*s are mutually independent and exponentially distributed with parameter* λ*, then*

$$\begin{aligned}\mathbb{E}[w_{t+1} - w_t] = \left(1 - \frac{1}{M}\right) \mathbb{E}[w_t - w_{t-1}] \\ - \frac{1}{M} \alpha \mathbb{E}\nabla_w f(w_t) \qquad (8)\end{aligned}$$

or equivalently, the asynchrony-induced (implicit) momentum is $\mu_S = 1 - \frac{1}{M}$*.*

The theorem suggests that, when training asynchronously, there are two sources of momentum:

- **Explicit or algorithmic momentum**: what we introduce algorithmically by tuning the momentum parameter in our optimizer.
- **Implicit or asynchrony-induced momentum**: what asynchrony contributes, as per Theorem 5.

For now we can consider that they act additively: the total effective momentum is the sum of explicit and implicit terms. This is a good first-order approximation, but can be improved by carefully modeling their higher-order interactions (cf. Section VI). The second theoretical prediction is that more workers introduce more momentum. Consider the following thought experiment, visualized in Figure 2. As we increase the number of asynchronous workers we tune, in each case, for the optimal explicit momentum.

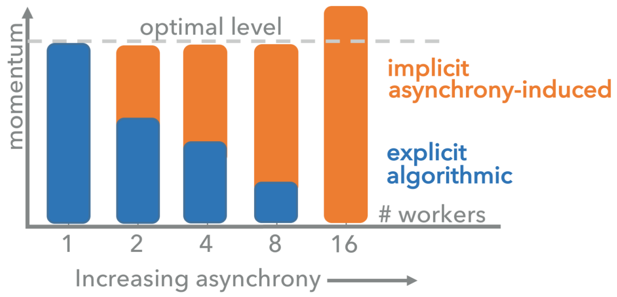

Fig. 2. Momentum behavior based on queuing model. Total momentum has some optimal value. When asynchrony-induced (implicit) momentum is less than that, we can algorithmically compensate for the rest. Beyond a certain point, asynchrony causes too much momentum, leading to statistical inefficiency.

This result gives some insight on the limits of asynchrony. Consider a case for which the optimal momentum in the sequential case is μ^*. Theorem 5 tells us that asynchrony-induced momentum is $\mu_S = 1 - 1/M$, for M asynchronous workers. Therefore there exists an M_0 such that $\mu_S > \mu^*$ for all $M > M_0$. In other words, too much asynchrony brings about too much momentum to the point of hurting performance. In Section VI, we show that there are ways to counteract these adverse effects of high asynchrony. In the next section, we validate this section's theoretical findings.

IV. Experimental Validation

We conduct experiments on 9 GPU machines on Amazon EC2 (g2.8xlarge). Each machine has 4 NVIDIA GRID K520 GPUs. One of these machines is used as a parameter server, and the other 8 machines can be organized into 1, 2, 4, 8 compute groups [22, 14]. Within each group, machines split a mini-batch and combine their updates synchronously. Cross-group updates are asynchronous. In this paper, we only study how momentum affects the number of iterations to reach a target loss for varying levels of asynchrony. Hence, groups are equivalent to the workers introduced in Section II. Our recent paper [14] includes end-to-end performance results; here we present results that pertain to our theory. We run experiments on two data sets: (i) CIFAR, and (ii) ImageNet. For CIFAR, we train the network provided by Caffe [23] and for ImageNet, we run CaffeNet [24]. For ImageNet, we grid search the explicit momentum in $\{0.0, 0.3, 0.6, 0.9\}$, learning rate in $\{0.1, 0.01, 0.001\}$ and set batch size to 256; for CIFAR we grid search the explicit momentum in $\{0.0, 0.1, ..., 0.9\}$, learning rate in $\{0.1, 0.01, \ldots, 0.00001\}$ and set batch size to 128.

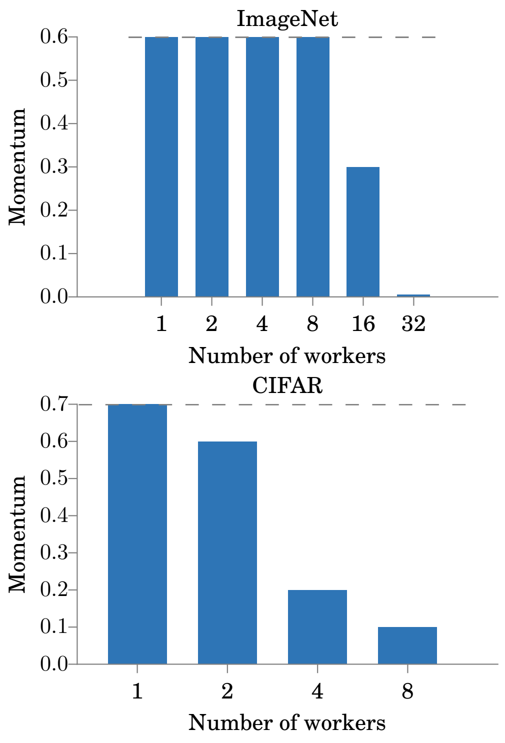

Fig. 3. Explicit momentum that achieves the best statistical efficiency. (Top) ImageNet. (Bottom) CIFAR.

Figure 3 shows the optimal amount of explicit momentum (the value that minimizes the number of steps to target loss) as we increase the number of workers. The top plot shows the results for ImageNet; the bottom plot shows CIFAR. We see that our theoretical prediction is supported by these measurements: the optimal explicit momentum decreases when we increase asynchrony.

We have established that the same interactions predicted in our theory manifest in our system, when we tune momentum. We now study the performance gains from this tuning process.

A. *Measuring Performance*

The costs and benefits of parallel optimization are best described using the notions of *hardware efficiency* and *statistical efficiency* [22]. The main performance metric is the wall-clock time to reach a certain training loss or accuracy. Given a fixed number of machines, we organize them into *compute groups* [22, 14], a "hybrid" between fully synchronous and fully asynchronous configurations. Within each group, machines combine their updates synchronously. Cross-group updates are asynchronous. We use this architecture in our experiments. In order to better understand how our design choices affect this metric we decompose it into two factors. The first factor, the number of steps to convergence, is mainly influenced by algorithmic choices and improvements. This factor leads to the notion of *statistical efficiency*. The second factor, the time to finish one step, is mainly influenced by hardware and system optimizations. It leads to the notion of *hardware efficiency*.

1) Hardware Efficiency: The obvious benefit of parallelization is "getting more done" in the same amount of time. In the case of SGD, we define hardware efficiency to be the relative time it takes to complete one step (mini-batch). Specifically, if using one compute group (fully synchronous setting) finishes a batch every T_1 seconds, and m groups finish a batch every T_m seconds, the hardware efficiency of using m groups is T_m/T_1.

2) Statistical Efficiency: On the other hand, some parallelization methods can have a detrimental effect on the quality of the achieved solution. In this case, asynchrony leads to staleness: some gradients are calculated using older models, w_t. Let I_m denote the number of steps required to reach some fixed loss, when using m groups. We define statistical efficiency as I_m/I_1. The product of hardware and statistical efficiency is the time it takes m groups to achieve the target accuracy normalized by the corresponding time for a single group; lower values mean faster performance.

By keeping the total number of workers fixed, we can use these measures of efficiency to study the tradeoffs between different configurations. Synchronous methods have better statistical efficiency, since all gradients have 0 staleness; they however suffer from worse hardware efficiency due to waiting at the synchronization barrier. Asynchronous methods provide worse statistical efficiency, but enjoy significant gains in terms of hardware efficiency: there are no stalls.

V. The importance of tuning

We see that tuning can significantly improve the statistical efficiency of asynchronous training. In Figure 4, we show results on CIFAR. We conduct experiments on 33 CPU machines on Amazon EC2 (c4.4xlarge). One of these machines is used as a parameter server, and the other machines are organized into compute groups. We grid search the explicit momentum in $\{-0.9, -0.675, -0.45, \ldots, 0.45, 0.675, 0.9\}$ and learning rate in $\{0.1, 0.01, 0.001, 0.0001, 0.00001\}$. We plot the normalized number of iterations for each configuration to reach a target loss, a statistical penalty compared to the best case. We first draw the penalty curve we get by using the standard value of momentum, 0.9, in all configurations. Then we draw the penalty curve we get when we grid-search momentum. We see that tuning momentum results into an improvement of about $2.5\times$ over the standard value $\mu_L = 0.9$.

Figure 5 shows the statistical penalty when tuning momentum on ImageNet. The cluster includes 8 machines, organized into compute groups and the setup is otherwise

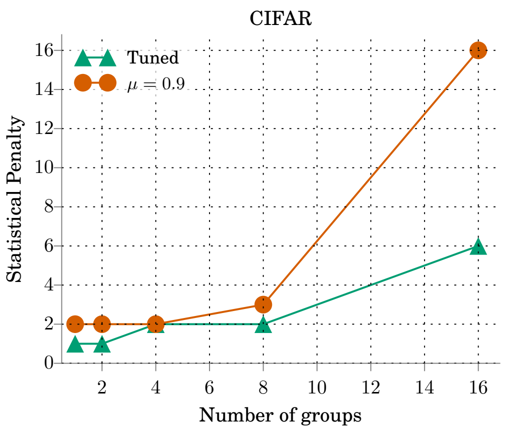

Fig. 4. The benefits of tuning momentum vs using a commonly prescribed value of $\mu_L = 0.9$ on the number of iterations to train the CIFAR dataset.

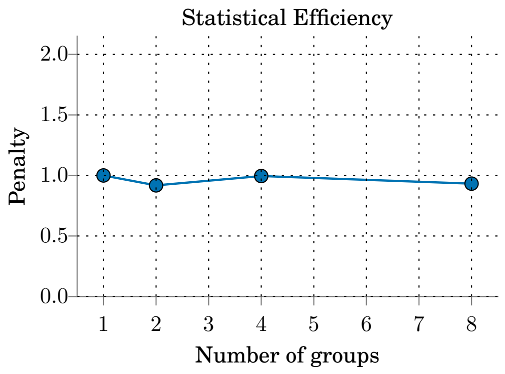

Fig. 5. Statistical efficiency for ImageNet dataset when tuning [14]. In this case, we pay no penalty for asynchrony.

the same as in Section IV. The statistical efficiency result was quite surprising to us: for ImageNet, even though workers perform dense updates on the model, there is *no* statistical efficiency penalty for up to 8 groups. This comes in contrast to standard results that say “asynchrony is fine as long as updates are sparse enough to prevent update collisions.” Two comments are in order: (i) this is in an intermediate stage of execution (not a cold start–which we have observed has different behavior–but 4000 iterations into the run), and (ii) there can be a penalty for some smaller models, or even at larger scales of workers. Our result provides some rough guidance for this behavior. Extensive experiments and detailed setup can be found in our systems paper [14].

VI. Counteracting the effects of asynchrony

In this section we take a closer look at the interaction between asynchrony and momentum. We derive an explicit update rule when both implicit and explicit momentum are present and prescribe a tuning strategy when all parameters are known. Perhaps surprisingly,

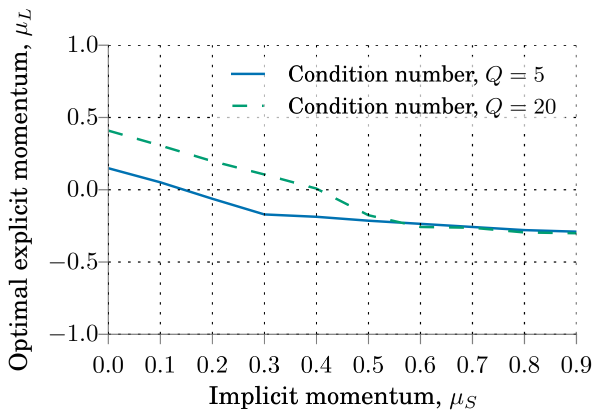

Fig. 6. Explicit momentum that yields the faster convergence for different values of implicit momentum, caused by staleness. For high implicit momentum, the optimal explicit momentum is negative.

using negative values of algorithmic momentum can—in some cases—improve the rate of convergence. Then we show experimentally that even when not all parameters are known, tuning via grid search can yield that negative values of algorithmic momentum are the most statistically efficient.

Let us assume the same staleness model of Assumption 4 in the presence of non-zero algorithmic momentum, μ_L.

Theorem 6. *Let the staleness distribution be geometric on* $\{0, 1, \ldots\}$ *with parameter* $1 - \mu_S$*, i.e.* $q_l = (1 - \mu_S)\mu_S^l$*. For* $\alpha' \triangleq (1 - \mu_S)\alpha$*, the expected update takes the momentum form of* (3).

$$\begin{aligned}\mathbb{E}[w_{t+1} - w_t] &= (\mu_L + \mu_S)\mathbb{E}[w_t - w_{t-1}] \\ &\quad - \mu_L\mu_S\mathbb{E}[w_{t-1} - w_{t-2}] - \alpha'\mathbb{E}\nabla_w f(w_t) \quad (9)\end{aligned}$$

Now we show that convergence rates for quadratic objectives can be easily computed numerically.

Theorem 7. *Consider a simple quadratic objective* $f(w) = \frac{1}{2}\|Aw - b\|_2^2$*, such that* $Aw_* = b$*. Let* λ_i *denote the* i*-th eigenvalue of* $A^\top A$ *and* t_i^* *denote the root of smallest magnitude for the polynomial*

$$g_i(t) = \mu_S\mu_L t^3 - (\mu_S + \mu_L + \mu_S\mu_L)\, t^2 + z_i t - 1, \quad (10)$$

where

$$z_i = 1 + \mu_S + \mu_L - \alpha(1 - \mu_S)\lambda_i. \quad (11)$$

The convergence rate of the expected iterates in the statement of Theorem 6 is given by

$$\|\mathbb{E}w_t - w_*\|_2 = O(\gamma^t), \qquad \text{where } \gamma \triangleq \max_i 1/|t_i^*|. \quad (12)$$

We can numerically evaluate the rates given in Theorem 7, and identify the values of explicit momentum that yield the fastest convergence for a given value of implicit momentum. Figure 6 shows the result of a fine

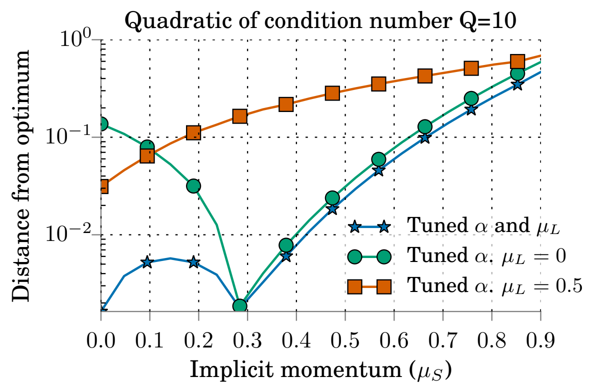

Fig. 7. Comparing momentum and step size tuning to step size tuning using rates from Theorem 7. For high values of μ_S, momentum tuning is faster than $\mu_L = 0.0$ because it selects negative momentum values.

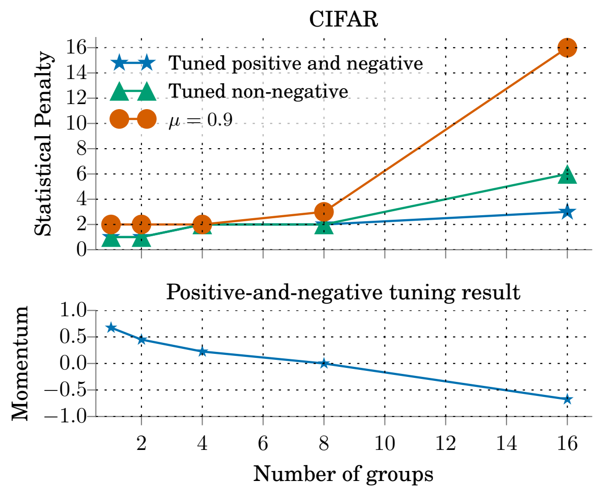

Fig. 8. Negative momentum tuning experiment on Omnivore and CIFAR. Allowing negative values improves further upon the performance we saw in Figure 4.

grid search over a range of values for explicit momentum μ_L and step size α, for quadratics of condition number $Q = 5, 20$. The figure shows that negative values of explicit momentum are optimal when staleness is high.

In Figure 7 we use the rates from Theorem 7 to simulate the result of 10 steps of the momentum algorithm on a quadratic of condition number 10.

We compare three tuning strategies: tuning step size and momentum, versus tuning step size only and fixing momentum to 0 or 0.5. We notice that proper tuning makes a difference for both low and high values of implicit momentum. In particular, the use of negative explicit momentum (for $\mu_S > 0.3$) results into a speedup of about $1.5x$ compared to using $\mu_L = 0.0$.

We test this idea on CIFAR on our system [14], in the setup described in Section V. Figure 8 shows the statistical penalty (increased number of iterations to the goal) we pay for asynchrony. The top curve is the penalty for using $\mu_L = 0.9$. The next one tunes momentum over non-negative values and achieves a speed-up of about $2.5\times$ using 16 groups. When we allow for tuning over negative momentum values, the penalty for 16 groups improves by another $2\times$. The bottom plot shows the momentum values selected by the latter tuning process. This provides experimental support to the numerical results Figure 6 and 7: negative momentum can reduce the statistical penalty further compared to non-negative tuning.

VII. Discussion and Future Work

We see that asynchrony-induced momentum complements algorithmic momentum. Asynchronous configurations run more efficiently with lower values of momentum. As a result, any performance evaluation of an asynchronous system needs to take this into account. A single, globally optimized value of explicit momentum used across configurations of varying asynchrony will yield an inaccurate evaluation: tuning is critical. Our results suggest simple ways to counteract the adverse statistical effects of asynchrony. A simple technique like negative momentum shows potential of pushing the limits of asynchrony further. We verified experimentally that the predicted behavior from our simplistic queueing model is present on real systems. Paraphrasing the maxim, any (noise) model is wrong but some are useful.

Our results can be turned into an optimizer that would tune explicit momentum based on current system statistics, like the measured distribution of staleness. The model presented here was simple and can be extended in many ways, which we plan to consider in future work.

- **Control-theory for momentum compensation** Ideas like negative momentum seem to work, but we can envision a disciplined way to deal with the adverse effects of asynchrony using control theoretic tools.
- **Topology.** We studied a simple NN topology. We can imagine interesting interactions between topology, physical mapping and queueing theory.
- **Data Sparsity and Irregular Access Patterns.** The work process can depend on the size of the support of the example used. Different applications involve data with different statistics; applications in Natural Language Processing often involve data following heavy-tailed distributions. Also, models with irregular access patterns like LSTMs [25] may give rise to different staleness distributions.
- **Optimization.** Different sparsity and staleness distributions naturally lead to momentum different to (3). Studying the convergence properties on momentum from arbitrary staleness could be of independent theoretical interest.

Acknowledgments

The authors would like to thank Chris De Sa for the discussion on asynchrony that was the precursor to the present work and Dan Iter for his thoughtful feedback. We would also like to thank Intel and Toshiba for support along with the Moore Foundation, DARPA through MEMEX, SIMPLEX, and XDATA programs, and the Office of Naval Research, under N000141410102.

References

[1] W. A. Gardner, "Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique," *Signal Processing*, vol. 6, no. 2, pp. 113–133, 1984.

[2] S.-i. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[3] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 116.

[4] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[5] F. Niu, B. Recht, C. Re, and S. Wright, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 571–582.

[8] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan, "Perturbed iterate analysis for asynchronous stochastic optimization," *arXiv preprint arXiv:1507.06970*, 2015.

[9] S. Chaturapruek, J. C. Duchi, and C. Ré, "Asynchronous stochastic convex optimization: the noise is in the noise and sgd don't care," in *NIPS*, 2015, pp. 1531–1539.

[10] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of hogwild-style algorithms," in *Advances in Neural Information Processing Systems*, 2015, pp. 2674–2682.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," *arXiv preprint arXiv:1605.08695*, 2016.

[12] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.

[13] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proc. of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 4.

[14] S. Hadjis, C. Zhang, I. Mitliagkas, and C. Ré, "Omnivore: An optimizer for multi-device deep learning on cpus and gpus," *arXiv preprint arXiv:1606.04487*, 2016.

[15] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[17] "Caffe solver documentation," http://caffe.berkeleyvision.org/tutorial/solver.html, accessed: 2016-09-29.

[18] S. Ruder, "An overview of gradient descent optimization algorithms," http://sebastianruder.com/optimizing-gradient-descent/index.html#momentum, accessed: 2016-09-29.

[19] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th international conference on machine learning (ICML-13)*, 2013, pp. 1139–1147.

[20] M. Hardt, B. Recht, and Y. Singer, "Train faster, generalize better: Stability of stochastic gradient descent," *arXiv preprint arXiv:1509.01240*, 2015.

[21] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, "Asynchrony begets momentum, with an application to deep learning," *arXiv preprint arXiv:1605.09774*, 2016.

[22] C. Zhang and C. Re, "Dimmwitted: A study of main-memory statistical analytics," *PVLDB*, vol. 7, no. 12, pp. 1283–1294, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1283-zhang.pdf

[23] "Caffe solver for CIFAR," https://github.com/BVLC/caffe/blob/master/examples/cifar10/cifar10_quick_solver.prototxt, accessed: 2016-09-28.

[24] "CaffeNet, solver for ImageNet," https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet, accessed: 2016-09-28.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

APPENDIX

A. Proof of Theorem 2

Proof. The statement follows by using (6) twice, and subtracting w_t from w_{t+1},

$$\begin{aligned} w_{t+1} - w_t =& w_t - w_{t-1} - \alpha\Big(\nabla_w f(v_t; z_{i_t}) \\ & - \nabla_w f(v_{t-1}; z_{i_{t-1}})\Big) \end{aligned}$$

rearranging terms and taking expectation with respect to the random selection of the $(i_t)_t$'s. This means we are not yet integrating over the randomness in the staleness variables, τ_t. Let $\mathcal{T}$ denote the smallest σ-algebra under which all the staleness variables are measurable. Then, using the independence in Assumption 1,

$$\begin{aligned} \mathbb{E}[w_{t+1} - w_t|\mathcal{T}] =& \mathbb{E}[w_t - w_{t-1}|\mathcal{T}] - \alpha\Big(\mathbb{E}[\nabla_w f(v_t)|\mathcal{T}] \\ & - \mathbb{E}[\nabla_w f(v_{t-1})|\mathcal{T}]\Big). \end{aligned}$$

Finally, integrating over all randomness,

$$\begin{aligned} \mathbb{E}[w_{t+1} - w_t] =& \mathbb{E}[w_t - w_{t-1}] \\ & - \alpha\left(\sum_{l=0}^{\infty} q_l \mathbb{E}\nabla_w f(w_{t-l}) - \sum_{l=0}^{\infty} q_l \mathbb{E}\nabla_w f(w_{t-l-1})\right) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha\Bigg(q_0 \mathbb{E}\nabla_w f(w_t) \\ & + \sum_{l=1}^{\infty} q_l \mathbb{E}\nabla_w f(w_{t-l}) - \sum_{l=0}^{\infty} q_l \mathbb{E}\nabla_w f(w_{t-l-1})\Bigg) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha q_0 \mathbb{E}\nabla_w f(w_t) \\ & - \alpha\Bigg(\sum_{l=0}^{\infty} q_{l+1} \mathbb{E}\nabla_w f(w_{t-l-1}) \\ & - \sum_{l=0}^{\infty} q_l \mathbb{E}\nabla_w f(w_{t-l-1})\Bigg) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha q_0 \mathbb{E}\nabla_w f(w_t) \\ & - \alpha \sum_{l=0}^{\infty} (q_{l+1} - q_l) \mathbb{E}\nabla_w f(w_{t-l-1}) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha q_0 \mathbb{E}\nabla_w f(w_t) \\ & + \alpha \sum_{l=0}^{\infty} (q_l - q_{l+1}) \mathbb{E}\nabla_w f(w_{t-l-1}) \end{aligned}$$

□

B. Proof of Theorem 3

Proof.

$$\begin{aligned} \mathbb{E}[w_{t+1} - w_t] =& \mathbb{E}[w_t - w_{t-1}] - \alpha q_0 \mathbb{E}\nabla_w f(w_t) \\ & + \alpha \sum_{l=0}^{\infty} (q_l - q_{l+1}) \mathbb{E}\nabla_w f(w_{t-l-1}) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha c \mathbb{E}\nabla_w f(w_t) \\ & + \alpha \sum_{l=0}^{\infty} \left(c\mu^l - c\mu^{l+1}\right) \mathbb{E}\nabla_w f(w_{t-l-1}) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha c \mathbb{E}\nabla_w f(w_t) \\ & + (1-\mu)\alpha \sum_{l=0}^{\infty} c\mu^l \mathbb{E}\nabla_w f(w_{t-l-1}) \\ =& \mathbb{E}[w_t - w_{t-1}] - \alpha c \mathbb{E}\nabla_w f(w_t) \\ & - (1-\mu)\mathbb{E}[w_t - w_{t-1}] \\ =& \mu\mathbb{E}[w_t - w_{t-1}] - \alpha c \mathbb{E}\nabla_w f(w_t) \\ =& \mu\mathbb{E}[w_t - w_{t-1}] - (1-\mu)\alpha \mathbb{E}\nabla_w f(w_t) \end{aligned}$$

□

C. Proof of Theorem 5

Proof. Let W_t denote the time the t-th iteration takes. Under Assumption 4, $W_t \sim Exp(\lambda)$. We want the staleness distribution τ_t: the number of writes in the time between the read and write of the reference worker in charge of step t. We call this random variable B_t. These in-between writes are performed by the remaining $M-1$ workers, and under Assumption 4, we get that in T units of time, the number of writes is

$$B_t(T) \sim \text{Poisson}(\lambda(M-1)T). \tag{13}$$

The staleness distribution is the number of writes by other workers in W_t units of time.

$$\tau_t \sim B_t(W_t) \tag{14}$$

It is a simple probability exercise[1] to show that τ_t is geometrically distributed on $\{0, 1, \ldots\}$.

$$\tau_t \sim \text{Geom}(p), \quad p = \frac{\lambda}{\lambda + (M-1)\lambda} = \frac{1}{M} \tag{15}$$

where M is the number of workers. Note that $\mathbb{E}\tau_t = M - 1$. Using this with Theorem 3 we get the statement.

$$\begin{aligned} \mathbb{E}[w_{t+1} - w_t] =& \left(1 - \frac{1}{M}\right)\mathbb{E}[w_t - w_{t-1}] \\ & - \frac{\alpha}{M}\mathbb{E}\nabla_w f(w_t) \end{aligned}$$

□

[1]For example, <9.7> in: http://www.stat.yale.edu/~pollard/Courses/241.fall97/Poisson.Proc.pdf

D. *Proof of Theorem 6*

Proof. We start with the update rule,

$$w_{t+1} = w_t - \alpha \nabla f(w_{t-\tau_t}) + \mu_L(w_t - w_{t-1}) \tag{16}$$

and consider the *expected step*,

$$\mathbb{E}w_{t+1} = \mathbb{E}w_t - \alpha \sum_{l=0}^{k} q_l \mathbb{E}\nabla f(w_{t-l}) + \mu_L \mathbb{E}[w_t - w_{t-1}]. \tag{17}$$

Staleness is geometrically distributed and independent for each write, i.e. $q_l = (1-\mu_S)\mu_S^l$ for $l \in \mathbb{N}_0$. We first rearrange the last equation

$$\mathbb{E}w_{t+1} - \mathbb{E}w_t - \mu_L \mathbb{E}[w_t - w_{t-1}] = -\alpha \sum_{l=0}^{k} (1-\mu_S)\mu_S^l \mathbb{E}\nabla f(w_{t-l}). \tag{18}$$

Now starting from (17),

$$\begin{aligned}
\mathbb{E}w_{t+1} =& \mathbb{E}w_t - \alpha \sum_{l=0}^{k} (1-\mu_S)\mu_S^l \mathbb{E}\nabla f(w_{t-l}) + \mu_L \mathbb{E}[w_t - w_{t-1}] \\
=& \mathbb{E}w_t - \alpha(1-\mu_S)\mathbb{E}\nabla f(w_t) - \alpha \sum_{l=1}^{k} (1-\mu_S)\mu_S^l \mathbb{E}\nabla f(w_{t-l}) + \mu_L \mathbb{E}[w_t - w_{t-1}] \\
=& \mathbb{E}w_t - \alpha(1-\mu_S)\mathbb{E}\nabla f(w_t) - \mu_S \alpha \sum_{l=0}^{k-1} (1-\mu_S)\mu_S^l \mathbb{E}\nabla f(w_{t-1-l}) + \mu_L \mathbb{E}[w_t - w_{t-1}]
\end{aligned}$$

and using (18),

$$\mathbb{E}w_{t+1} = \mathbb{E}w_t - \alpha(1-\mu_S)\mathbb{E}\nabla f(w_t) + \mu_S\big(\mathbb{E}w_t - \mathbb{E}w_{t-1} - \mu_L \mathbb{E}[w_{t-1} - w_{t-2}]\big) + \mu_L \mathbb{E}[w_t - w_{t-1}]$$

and finally,

$$\mathbb{E}w_{t+1} = (1+\mu_S+\mu_L)\mathbb{E}w_t - \alpha(1-\mu_S)\mathbb{E}\nabla f(w_t) - (\mu_S+\mu_L+\mu_S\mu_L)\,\mathbb{E}w_{t-1} + \mu_S\mu_L \mathbb{E}w_{t-2} \tag{19}$$

This recurrence holds for $k \geq 2$. □

E. *Proof of Theorem 7*

Proof. We will use the polynomial family $q_k(z)$ to describe the behavior of $\mathbb{E}[w_k - w_*]$. From $Aw_* = b$, we get $\nabla f(w) = A^\top A(w - w_*)$. Then, the statement of Theorem 6, can be equivalently described by the following polynomial recursion.

$$q_{k+1}(z) = zq_k(z) - (\mu_S+\mu_L+\mu_S\mu_L)\, q_{k-1}(z) + \mu_S\mu_L q_{k-2}(z) \tag{20}$$

for and $k \geq 2$, where $z = (1+\mu_S+\mu_L)I - \alpha(1-\mu_S)A^T A$. Note that $q_0(z) = 1$, $q_1(z) = z - \mu_L - \mu_S$ and $q_2(z) = z^2 - (\mu_L+\mu_S)z - (\mu_L+\mu_S)$. To get the generating function for this recurrence, we multiply by t^{k+1} and sum over $k = 2, \ldots \infty$.

$$\sum_{k=2}^{\infty} q_{k+1}(z)t^{k+1} = \sum_{k=2}^{\infty} zy_k(z)t^{k+1} - \sum_{k=2}^{\infty} (\mu_S+\mu_L+\mu_S\mu_L)\, y_{k-1}(z)t^{k+1} + \mu_S\mu_L \sum_{k=2}^{\infty} w_{t-2}(z)t^{k+1}$$

$$G(z) - q_2(z) - q_1(z) - q_0(z) = zt(G(z) - q_1(z) - q_0(z)) - (\mu_S+\mu_L+\mu_S\mu_L)\, t^2(G(z) - q_0(z)) + \mu_S\mu_L t^2 G(z)$$

Rearranging,

$$\begin{aligned}
&G(z)\left(1 - zt + (\mu_S+\mu_L+\mu_S\mu_L)\, t^2 - \mu_S\mu_L t^3\right) \\
&\quad= q_2(z) + q_1(z) + q_0(z) - zt(q_1(z) + q_0(z)) + (\mu_S+\mu_L+\mu_S\mu_L)\, t^2 q_0(z) \\
&\quad= q_2(z) + q_1(z) + 1 - zt(q_1(z) + 1) + (\mu_S+\mu_L+\mu_S\mu_L)\, t^2 \\
&\quad= z^2 - (\mu_L+\mu_S)z - (\mu_L+\mu_S) + z - \mu_L - \mu_S + 1 - zt(z - \mu_L - \mu_S + 1) + (\mu_S+\mu_L+\mu_S\mu_L)\, t^2 \\
&\quad= [\mu_S+\mu_L+\mu_S\mu_L]\, t^2 - [z(z-\mu_L-\mu_S+1)]\, t + \left[z^2 + (1-\mu_L-\mu_S)z + 1 - 2(\mu_L+\mu_S)\right]
\end{aligned}$$

$$G(z) = -\frac{[\mu_S + \mu_L + \mu_S\mu_L]\, t^2 - [z(z - \mu_L - \mu_S + 1)]\, t + \left[z^2 + (1 - \mu_L - \mu_S)z + 1 - 2(\mu_L + \mu_S)\right]}{\mu_S\mu_L t^3 - (\mu_S + \mu_L + \mu_S\mu_L)\, t^2 + zt - 1}$$

The roots of the denominator (the growth polynomial) of the generating function, dictate the rate of convergence. In particular, the inverse of the largest root magnitude gives us the desired rate. Let $A^\top A = Q\Lambda Q^T$ be the eigendecomposition of $A^T A$, with eigenvalues λ_i. Let $v_t = Q^T w_t$ and note that for every i we get a scalar recurrence for $v_t(i)$.

$$\mathbb{E}v_{t+1}(i) = (1 + \mu_S + \mu_L)\mathbb{E}v_t(i) - \alpha(1 - \mu_S)\lambda_i(v_t(i) - v_*(i)) - (\mu_S + \mu_L + \mu_S\mu_L)\, \mathbb{E}v_{t-1}(i) + \mu_S\mu_L \mathbb{E}v_{t-2}(i) \tag{21}$$

According to the analysis above, its growth polynomial is

$$g_i(t) = \mu_S\mu_L t^3 - (\mu_S + \mu_L + \mu_S\mu_L)\, t^2 + z_i t - 1, \tag{22}$$

where

$$z_i = 1 + \mu_S + \mu_L - \alpha(1 - \mu_S)\lambda_i. \tag{23}$$

Now let t_i^* denote the root of smallest magnitude for $g_i(t)$. The rate of convergence along the i eigendirection is $\gamma_i = O(1/|t_i^*|)$. The rate for w_t is dominated by the largest γ_i, which yields the statement. $\square$