# REVISITING DISTRIBUTED SYNCHRONOUS SGD

**Jianmin Chen**[*], **Xinghao Pan**[*†], **Rajat Monga, Samy Bengio**
Google Brain
Mountain View, CA, USA
{jmchen,xinghao,rajatmonga,bengio}@google.com

**Rafal Jozefowicz**
OpenAI
San Francisco, CA, USA
rafal@openai.com

## ABSTRACT

Distributed training of deep learning models on large-scale training data is typically conducted with *asynchronous* stochastic optimization to maximize the rate of updates, at the cost of additional noise introduced from asynchrony. In contrast, the *synchronous* approach is often thought to be impractical due to idle time wasted on waiting for straggling workers. We revisit these conventional beliefs in this paper, and examine the weaknesses of both approaches. We demonstrate that a third approach, synchronous optimization with backup workers, can avoid asynchronous noise while mitigating for the worst stragglers. Our approach is empirically validated and shown to converge *faster* and to *better* test accuracies.

## 1 INTRODUCTION

The recent success of deep learning approaches for domains like speech recognition (Hinton et al., 2012) and computer vision (Ioffe & Szegedy, 2015) stems from many algorithmic improvements but also from the fact that the size of available training data has grown significantly over the years, together with the computing power, in terms of both CPUs and GPUs. While a single GPU often provides algorithmic simplicity and speed up to a given scale of data and model, there exist an operating point where a distributed implementation of training algorithms for deep architectures becomes necessary.

Currently, popular distributed training algorithms include mini-batch versions of stochastic gradient descent (SGD) and other stochastic optimization algorithms such as AdaGrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012), and ADAM (Kingma & Ba, 2014). Unfortunately, bulk-synchronous implementations of stochastic optimization are often slow in practice due to the need to wait for the slowest machine in each synchronous batch. To circumvent this problem, practitioners have resorted to asynchronous approaches which emphasize speed by using potentially stale information for computation. While asynchronous training have proven to be faster than their synchronous counterparts, they often result in convergence to poorer results.

In this paper[1], we revisit synchronous learning, and propose a method for mitigating stragglers in synchronous stochastic optimization. Specifically, we synchronously compute a mini-batch gradient with only a subset of worker machines, thus alleviating the straggler effect while avoiding any staleness in our gradients. The primary contributions of our paper are:

- Illustration of how gradient staleness in asynchronous training negatively impacts test accuracy and is exacerbated by deep models.
- Measurement of machine response times for synchronous stochastic optimization in a large deployment of 100 GPUs, showing how stragglers in the tail end affect convergence speed.
- Proposal of synchronous stochastic optimization with *backup workers* to mitigate straggler effects without gradient staleness.
- Establishing the need to measure both speed of convergence and test accuracy of optimum for empirical validation.

[*] Joint first authors

[†] UC Berkeley, Berkeley, CA, USA, xinghao@eecs.berkeley.edu

[1] This is an extension of our ICLR 2016 workshop extended abstract (Chen et al., 2016).

- Empirical demonstration that our proposed synchronous training method outperforms asynchronous training by converging faster and to better test accuracies.

The remainder of this paper is organized as follows. We briefly present preliminaries and notation in Section 1.1. Section 2 describes asynchronous stochastic optimization and presents experimental evidence of gradient staleness in deep neural network models. We present our approach in Section 3, and exhibit straggler effects that motivate the approach. We then empirically evaluate our approach in Sections 4. Related work is discussed in Section 5, and we conclude in Section 6.

### 1.1 Preliminaries and Notation

Given a dataset $\mathcal{X} = \{x_i : i = 1, \ldots, |\mathcal{X}|\}$, our goal is to learn the parameters $\theta$ of a model with respect to an empirical loss function $f$, defined as $f(\theta) \triangleq \frac{1}{|\mathcal{X}|} \sum_{i=1}^{|\mathcal{X}|} F(x_i; \theta)$, where $F(x_i; \theta)$ is the loss with respect to a datapoint $x_i$ and the model $\theta$.

A first-order stochastic optimization algorithm achieves this by iteratively updating $\theta$ using a stochastic gradient $G \triangleq \nabla F(x_i; \theta)$ computed at a randomly sampled $x_i$, producing a sequence of models $\theta^{(0)}, \theta^{(1)}, \ldots$. Stochastic optimization algorithms differ in their update equations. For example, the update of SGD is $\theta^{(t+1)} = \theta^{(t)} - \gamma_t G^{(t)} = \theta^{(t)} - \gamma_t \nabla F(x_i; \theta^{(t)})$, where $\gamma_t$ is the *learning rate* or *step size* at iteration $t$. A mini-batch version of the stochastic optimization algorithm computes the stochastic gradient over mini-batch of size $B$ instead of a single datapoint, i.e., $G \triangleq \frac{1}{B} \sum_{i=1}^{B} \nabla F(\widetilde{x}_i; \theta^{(t)})$, where $\widetilde{x}_i$'s are randomly sampled from $\mathcal{X}$. We will often evaluate performance on an exponential moving average $\bar{\theta}^{(t)} = \alpha \bar{\theta}^{(t-1)} + (1 - \alpha)\theta^{(t)}$ with decay rate $\alpha$.

Our interest is in *distributed* stochastic optimization using $N$ worker machines in charge of computing stochastic gradients that are sent to $M$ parameter servers. Each parameter server $j$ is responsible for storing a subset $\theta[j]$ of the model, and performing updates on $\theta[j]$. In the synchronous setting, we will also introduce additional $b$ *backup workers* for straggler mitigation.

## 2 Asynchronous Stochastic Optimization

An approach for a distributed stochastic gradient descent algorithm was presented in Dean et al. (2012), consisting of two main ingredients. First, the parameters of the model are distributed on multiple servers, depending on the architecture. This set of servers are called the *parameter servers*. Second, there can be multiple workers processing data in parallel and communicating with the parameter servers. Each worker processes a mini-batch of data independently of the others, as follows:

- The worker fetches from the parameter servers the most up-to-date parameters of the model needed to process the current mini-batch;
- It then computes gradients of the loss with respect to these parameters;
- Finally, these gradients are sent back to the parameter servers, which then updates the model accordingly.

Since each worker communicates with the parameter servers independently of the others, this is called *Asynchronous Stochastic Gradient Descent* (Async-SGD), or more generally, *Asynchronous Stochastic Optimization* (Async-Opt). A similar approach was later proposed by Chilimbi et al. (2014). Async-Opt is presented in Algorithms 1 and 2.

In practice, the updates of Async-Opt are different than those of serially running the stochastic optimization algorithm for two reasons. Firstly, the read operation (Algo 1 Line 2) on a worker may be interleaved with updates by other workers to different parameter servers, so the resultant $\widehat{\theta}_k$ may not be consistent with any parameter incarnation $\theta^{(t)}$. Secondly, model updates may have occurred while a worker is computing its stochastic gradient; hence, the resultant gradients are typically computed with respect to outdated parameters. We refer to these as *stale* gradients, and its *staleness* as the number of updates that have occurred between its corresponding read and update operations.

Understanding the theoretical impact of staleness is difficult work and the topic of many recent papers, e.g. Recht et al. (2011); Duchi et al. (2013); Leblond et al. (2016); Reddi et al. (2015);

**Algorithm 1:** Async-SGD worker $k$

**Input**: Dataset $\mathcal{X}$
**Input**: $B$ mini-batch size
**while** *True* **do**
  Read $\widehat{\theta}_k = (\theta[0], \ldots, \theta[M])$ from PSs.
  $G_k^{(t)} := 0$.
  **for** $i = 1, \ldots, B$ **do**
    Sample datapoint $\widetilde{x}_i$ from $\mathcal{X}$.
    $G_k^{(t)} \leftarrow G_k^{(t)} + \frac{1}{B}\nabla F(\widetilde{x}_i; \widehat{\theta}_k)$.
  **end**
  Send $G_k^{(t)}$ to parameter servers.
**end**

**Algorithm 2:** Async-SGD Parameter Server $j$

**Input**: $\gamma_0, \gamma_1, \ldots$ learning rates.
**Input**: $\alpha$ decay rate.
**Input**: $\theta^{(0)}$ model initialization.
**for** $t = 0, 1, \ldots$ **do**
  Wait for gradient $G$ from any worker.
  $\theta^{(t+1)}[j] \leftarrow \theta^{(t)}[j] - \gamma_t G[j]$.
  $\bar{\theta}^{(t)}[j] = \alpha\bar{\theta}^{(t-1)}[j] + (1-\alpha)\theta^{(t)}[j]$.
**end**

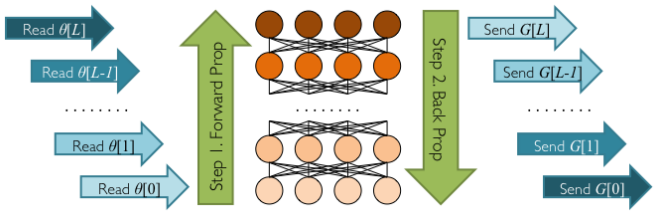



Figure 1: Gradient staleness dependence on model layer. Gradients are computed in a bottom-up forward propagation step followed by a top-down back propagation step. Parameters are read from servers in the forward prop, but gradients are sent to servers during the back prop. Thus, gradients of lower layers are more stale than top layers.

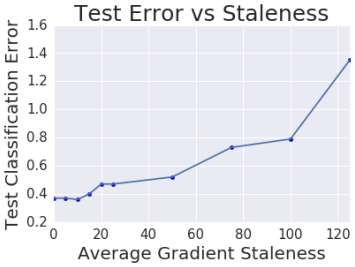



Figure 2: Degradation of test classification error with increasing average gradient staleness in MNIST CNN model.

De Sa et al. (2015); Mania et al. (2015), most of which focus on individual algorithms, under strong assumptions that may not hold up in practice. This is further complicated by deep models with multiple layers, since the times at which model parameters are read and which gradients are computed and sent are dependent on the depth of the layers (Figure 1). To better understand this dependence in real models, we collected staleness statistics on a Async-Opt run with 40 workers on a 18-layer Inception model (Szegedy et al., 2016) trained on the ImageNet Challenge dataset (Russakovsky et al., 2015), as shown in Table 1.

Despite the abovementioned problems, Async-Opt has been shown to be scale well up to a few dozens of workers for some models. However, at larger scales, increasing the number of machines (and thus staleness of gradients) can result in poorer trained models.

| Layer | Min | Mean | Median | Max | Std Dev | Count |
|---|---|---|---|---|---|---|
| 18 | 4 | 14.54 | 13.94 | 29 | 3.83 | 10908 |
| 12 | 5 | 11.35 | 11.3 | 23 | 3.09 | 44478 |
| 11 | 8 | 19.8 | 19.59 | 34 | 3.65 | 187 |
| 0 | 24 | 38.97 | 38.43 | 61 | 5.43 | 178 |

Table 1: Staleness of gradients in a 18-layer Inception model. Gradients were collected in a run of asynchronous training using 40 machines. *Staleness* of a gradient is measured as the number of updates that have occurred between its corresponding read and update operations. The staleness of gradients increases from a mean of ∼14.5 in the top layer (Layer 18) to ∼39.0 in the bottom layer (Layer 0).

### 2.1 Impact of staleness on test accuracy

We explore how increased staleness contributes to training of poorer models. In order to mimic the setting on a smaller scale, we trained a state-of-the-art MNIST CNN model but simulated staleness by using old gradients for the parameter updates. Details of the model and training are provided in Appendix A.1.

The best final classification error on a test set was 0.36%, which increases to 0.47% with average gradient staleness of 20 steps, and up to 0.79% with 50 steps (see Figure 2).

Once the average simulated staleness was chosen to be more than 15 steps, the results started to significantly deteriorate and the training itself became much less stable. We had to employ following tricks to prevent the results from blowing up:

- Slowly increase the staleness over the first 3 epochs of training. This mimics increasing the number of asynchronous workers and is also very important in practice for some of the models we experimented with (e.g. large word-level language models). The trick was not relevant with a simulated staleness less than 15 but became crucial for larger values.
- Use lower initial learning rates when staleness is at least 20, which reduces a frequency of explosions (train error goes to 90%). This observation is similar to what we found in other experiments - we were able to use much larger learning rates with synchronous training and the results were also more stable.
- Even with above tricks the divergence occurs occasionally and we found that restarting training from random weights can lead to more successful runs. The best results were then chosen based on validation set performance.

## 3 Revisting Synchronous Stochastic Optimization

Both Dean et al. (2012) and Chilimbi et al. (2014) use versions of Async-SGD where the main potential problem is that each worker computes gradients over a potentially old version of the model. In order to remove this discrepancy, we propose here to reconsider a synchronous version of distributed stochastic gradient descent (Sync-SGD), or more generally, *Synchronous Stochastic Optimization* (Sync-Opt), where the parameter servers wait for all workers to send their gradients, aggregate them, and send the updated parameters to all workers afterward. This ensures that the actual algorithm is a true mini-batch stochastic gradient descent, with an effective batch size equal to the sum of all the mini-batch sizes of the workers.

While this approach solves the staleness problem, it also introduces the potential problem that the actual update time now depends on the slowest worker. Although workers have equivalent computation and network communication workload, slow stragglers may result from failing hardware, or contention on shared underlying hardware resources in data centers, or even due to preemption by other jobs.

To alleviate the straggler problem, we introduce *backup workers* (Dean & Barroso, 2013) as follows: instead of having only $N$ workers, we add $b$ extra workers, but as soon as the parameter servers receive gradients from any $N$ workers, they stop waiting and update their parameters using the $N$ gradients. The slowest $b$ workers' gradients will be dropped when they arrive. Our method is presented in Algorithms 3, 4.

**Algorithm 3:** Sync-SGD worker $k$, where $k = 1, \ldots, N + b$

**Input**: Dataset $\mathcal{X}$
**Input**: $B$ mini-batch size
**for** $t = 0, 1, \ldots$ **do**
  Wait to read $\theta^{(t)} = (\theta^{(t)}[0], \ldots, \theta^{(t)}[M])$ from parameter servers.
  $G_k^{(t)} := 0$
  **for** $i = 1, \ldots, B$ **do**
    Sample datapoint $\widetilde{x}_{k,i}$ from $\mathcal{X}$.
    $G_k^{(t)} \leftarrow G_k^{(t)} + \frac{1}{B} \nabla F(\widetilde{x}_{k,i}; \theta^{(t)})$.
  **end**
  Send $(G_k^{(t)}, t)$ to parameter servers.
**end**

**Algorithm 4:** Sync-SGD Parameter Server $j$

**Input**: $\gamma_0, \gamma_1, \ldots$ learning rates.
**Input**: $\alpha$ decay rate.
**Input**: $N$ number of mini-batches to aggregate.
**Input**: $\theta^{(0)}$ model initialization.
**for** $t = 0, 1, \ldots$ **do**
  $\mathcal{G} = \{\}$
  **while** $|\mathcal{G}| < N$ **do**
    Wait for $(G, t')$ from any worker.
    **if** $t' == t$ **then** $\mathcal{G} \leftarrow \mathcal{G} \cup \{G\}$.
    **else** Drop gradient $G$.
  **end**
  $\theta^{(t+1)}[j] \leftarrow \theta^{(t)}[j] - \frac{\gamma_t}{N} \sum_{G \in \mathcal{G}} G[j]$.
  $\bar{\theta}^{(t)}[j] = \alpha \bar{\theta}^{(t-1)}[j] + (1 - \alpha) \theta^{(t)}[j]$.
**end**

### 3.1 STRAGGLER EFFECTS

The use of backup workers is motivated by the need to mitigate slow stragglers while maximizing computation. We investigate the effect of stragglers on Sync-Opt model training here.

We ran Sync-Opt with $N = 100$ workers, $b = 0$ backups, and 19 parameter servers on the Inception model. Using one variable as a proxy, we collected for each iteration both the start time of the iteration and the time when the $k$th gradient of that variable arrived at the parameter server. These times are presented in Figure 3 for $k = 1, 50, 90, 97, 98, 99, 100$. Note that 80% of the 98th gradient arrives in under 2s, whereas only 30% of the final gradient do. Furthermore, the time to collect the final few gradients grows exponentially, resulting in wasted idle resources and time expended to wait for the slowest gradients. This exponential increase is also seen in Figure 4.

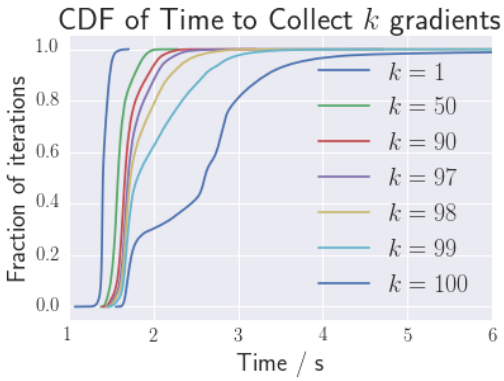



Figure 3: CDF of time taken to aggregate gradients from $N$ machines. For clarity, we only show times of $\leq$ 6s; the maximum observed time is 310s.

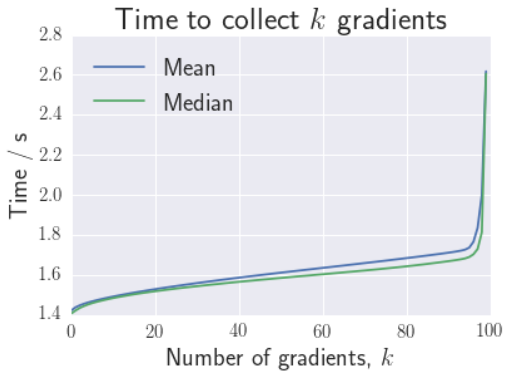



Figure 4: Mean and median times, across all iterations, to collect $k$ gradients on $N = 100$ workers and $b = 0$ backups. Most mean times fall between 1.4s and 1.8s, except of final few gradients.

Thus, one might choose to drop slow stragglers to decrease the iteration time. However, using fewer machines implies a smaller effective mini-batch size and thus greater gradient variance, which in turn could require more iterations for convergence. We examine this relationship by running Sync-Opt[2] with $N = 50, 70, 80, 90, 100$ and $b = 6$, and note the number of iterations required for convergence in Figure 5. Additional details of this training are provided in Appendix A.2. As $N$ is doubled from 50 to 100, the number of iterations to converge nearly halves from $137.5e3$ to $76.2e3$.

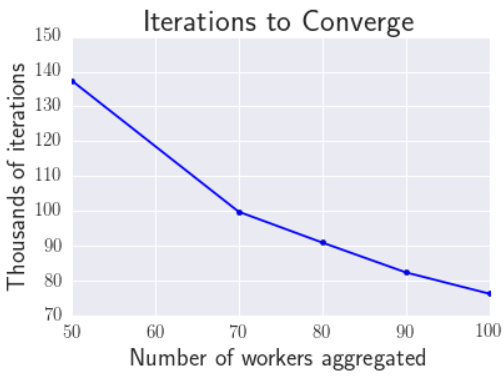



Figure 5: Number of iterations to converge when aggregating gradient from $N$ machines.

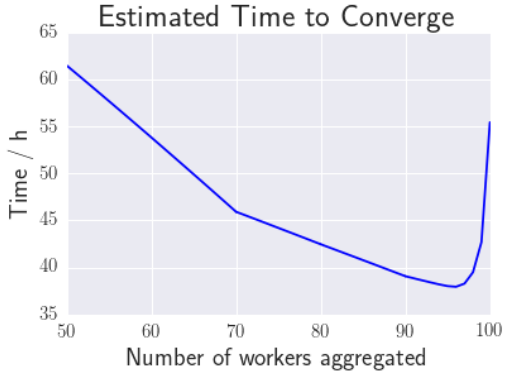



Figure 6: Estimated time to converge when aggregating gradients from $N$ machines on a $N + b = 100$ machine configuration. Convergence is fastest when choosing $N = 96$, $b = 4$.

[2] Since we are interested in the gradient quality and convergence behavior but *not* running time in this experiment, the backups serve only to reduce our data collection time but do not affect our analysis.

Hence, there is a trade-off between dropping more stragglers to reduce iteration time, and waiting for more gradients to improve the gradient quality. Consider a hypothetical setting where we have $N + b = 100$ machines, and we wish to choose the best configuration of $N$ and $b$ to minimize running time to convergence[3]. For each configuration, we can estimate the iterations required from Figure 5 (linearly interpolating for values of $N$ for which we did not collect data). We can multiply this with the mean iteration times (Figure 4) to obtain the *running time* required to converge for each setting of $N$ and $b$. These results are shown in Figure 6, indicating that $N = 96$, $b = 4$ converges fastest. Therefore, this motivates our choice to use a few backup workers for mitigating stragglers.

# 4 EXPERIMENTS

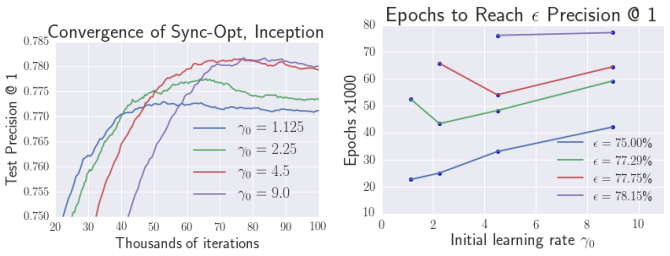
In this section, we present our empirical comparisons of synchronous and asynchronous distributed stochastic optimization algorithms as applied to models such as Inception and PixelCNN. All experiments in this paper are using the TensorFlow system (Abadi et al., 2015).

## 4.1 METRICS OF COMPARISON: FASTER CONVERGENCE, BETTER OPTIMUM

We are interested in two metrics of comparison for our empirical validation: (1) test error or accuracy, and (2) speed of convergence[3]. We point out that for non-convex deep learning models, it is possible to converge faster to a poorer local optimum. Here we show a simple example with Inception using different learning rates.

| Initial rate $\gamma_0$ | Test precision at convergence | Epochs to converge |
|---|---|---|
| 1.125 | 77.29% | 52628 |
| 2.25 | 77.75% | 65811 |
| 4.5 | 78.15% | 76209 |
| 9.0 | 78.17% | 77235 |

Table 2: Test accuracies at convergence and number of epochs to converge for different initial learning rates $\gamma_0$. Low initial learning rates result in faster convergence to poorer local optimum.




(a) Convergence (b) Epochs to $\epsilon$ test precision 1.

Figure 7: Convergence of Sync-Opt on Inception model using $N = 100$ workers and $b = 6$ backups, with varying initial learning rates $\gamma_0$. To reach a lower $\epsilon$ test precision, small $\gamma_0$'s require fewer epochs than large $\gamma_0$'s. However, small $\gamma_0$'s either fail to attain high $\epsilon$ precision, or take more epochs than higher $\gamma_0$'s.

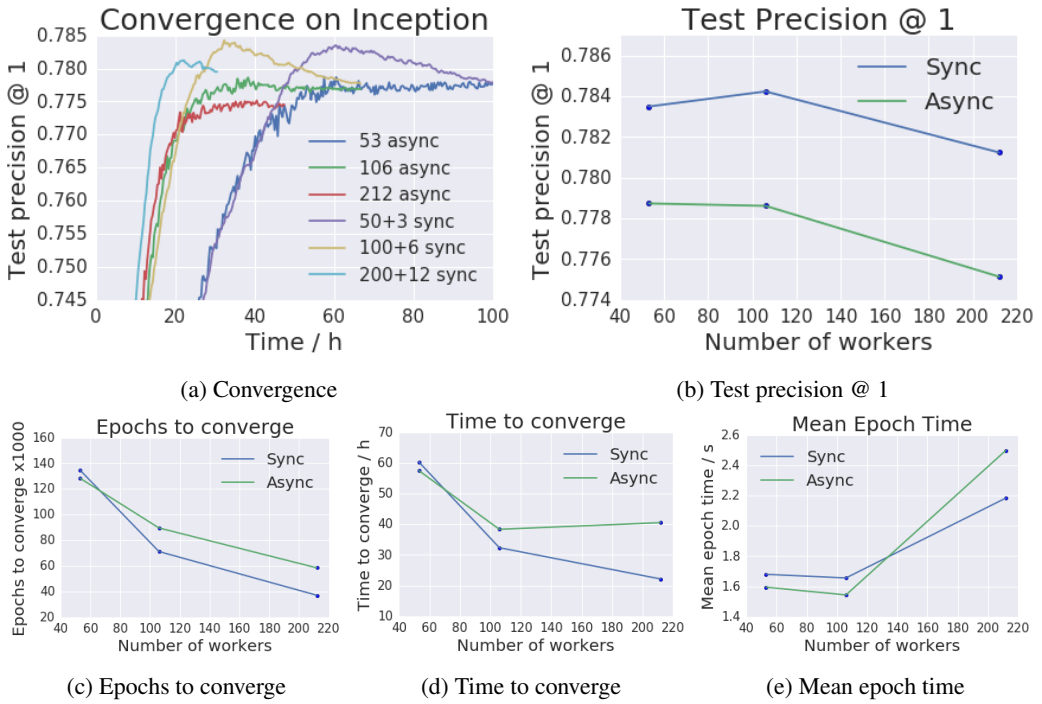
We ran Sync-Opt on Inception with $N = 100$ and $b = 6$, but varied the initial learning rate $\gamma_0$ between 1.125 and 9.0. (Learning rates are exponentially decreased with iterations.) Table 2 shows that smaller $\gamma_0$ converge faster, but to poorer test precisions. Focusing on speed on an early phase of training could lead to misleading conclusions if we fail to account for eventual convergence. For example, Figure 3b shows that using $\gamma_0 = 1.125$ reaches $\epsilon = 75\%$ precision $1.5\times$ faster than $\gamma_0 = 4.5$, but is slower for $\epsilon = 77.75\%$, and fails to reach higher precisions.

## 4.2 INCEPTION

We conducted experiments on the Inception model (Szegedy et al., 2016) trained on ImageNet Challenge dataset (Russakovsky et al., 2015), where the task is to classify images out of 1000 categories. We used several configurations, varying $N + b$ from 53 to 212 workers. Additional details of the training are provided in Appendix A.3. An *epoch* is a synchronous iteration for Sync-Opt, or a full pass of $N$ updates for Async-Opt, which represent similar amounts of computation. Results of this experiment are presented in Figure 8.

Figure 8b shows that Sync-Opt outperforms Async-Opt in test precision: Sync-Opt attains ~0.5% better test precision than Async-Opt for comparable $N + b$ workers. Furthermore, Sync-Opt con-

[3]Convergence is defined as the point where maximum test accuracy or lowest test error is reached.




(a) Convergence

(b) Test precision @ 1

(c) Epochs to converge

(d) Time to converge

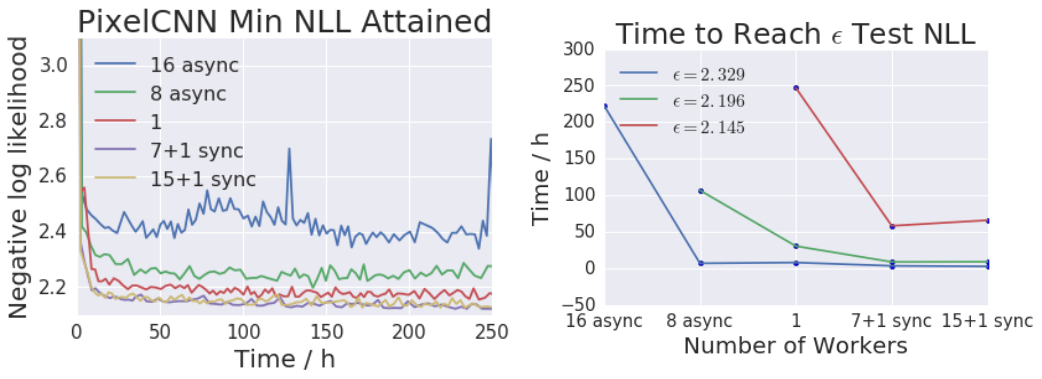
(e) Mean epoch time

Figure 8: Convergence of Sync-Opt and Async-Opt on Inception model using varying number of machines. Sync-Opt with backup workers converge faster, with fewer epochs, to higher test accuracies.

verges 6h and 18h faster than Async-Opt for 106 and 212 workers respectively, and is 3h slower when 53 workers are used, as seen in Figure 8d. This difference in speed is largely due to the fewer epochs (Figure 8c) needed by Sync-Opt, but comparable or better epoch time (Figure 8e).

### 4.3 PIXELCNN EXPERIMENTS

The second model we experimented on is PixelCNN (Oord et al., 2016), a conditional image generation deep neural network, which we train on the CIFAR-10 (Krizhevsky & Hinton, 2009) dataset. Configurations of $N + b = 1, 8, 16$ workers were used; for Sync-Opt, we always used $b = 1$ backup worker. Additional details are provided in Appendix A.4.




(a)

(b)

Figure 9: Convergence of synchronous and asynchronous training on PixelCNN model. Sync-Opt achieves lower negative log likelihood in less time than Async-Opt.

Convergence of the test negative log likelihood (NLL) on PixelCNN is shown in Figure 9a, where lower is better. Observe that Sync-Opt obtains lower NLL than Async-Opt; in fact, Async-Opt is even outperformed by serial RMSProp with $N = 1$ worker, with degrading performance as $N$ increases from 8 to 16. Figure 9b further shows the time taken to reach $\epsilon$ test NLL. Sync-Opt reduces the time to reach $\epsilon = 2.145$ from 247h to 58.3h; this NLL is not even achieved by Async-Opt.

## 5 Related Work

Multicore and distributed optimization algorithms have received much attention in recent years. Asynchronous algorithms include Recht et al. (2011); Duchi et al. (2013); Zhang et al. (2015a); Reddi et al. (2015); Leblond et al. (2016). Implementations of asynchronous optimization include Xing et al. (2015); Li et al. (2014); Chilimbi et al. (2014). Attempts have also been made in Zinkevich et al. (2010) and Zhang & Jordan (2015) to algorithmically improve synchronous SGD.

An alternative solution, "softsync", was presented in Zhang et al. (2015b), which proposed batching gradients from multiple machines before performing an asynchronous SGD update, thereby reducing the effective staleness of gradients. Similar to our proposal, softsync avoids stragglers by not forcing updates to wait for the slowest worker. However, softsync allows the use of stale gradients but we do not. The two solutions provide different explorations of the trade-off between high accuracy (by minimizing staleness) and fast throughput (by avoiding stragglers).

Watcharapichat et al. (2016) introduces a distributed deep learning system without parameter servers, by having workers interleave gradient computation and communication in a round-robin pattern. Like Async-Opt, this approach suffers from staleness. We also note that in principle, workers in Sync-Opt can double as parameter servers and execute the update operations and avoid the need to partition hardware resources between workers and servers.

Das et al. (2016) analyzes distributed stochastic optimization and optimizes the system by solving detailed system balance equations. We believe this approach is complimentary to our work, and could potentially be applied to guide the choice of systems configurations for Sync-Opt.

Keskar et al. (2016) suggests that large batch sizes for synchronous stochastic optimization leads to poorer generalization. Our effective batch size increases linearly with the number of workers $N$. However, we did not observe this effect in our experiments; we believe we are not yet in the large batch size regime examined by Keskar et al. (2016).

## 6 Conclusion and Future Work

Distributed training strategies for deep learning architectures will become ever more important as the size of datasets increases. In this work, we have shown how both synchronous and asynchronous distributed stochastic optimization suffer from their respective weaknesses of stragglers and staleness. This has motivated our development of synchronous stochastic optimization with backup workers, which we show to be a viable and scalable strategy.

We are currently experimenting with different kinds of datasets, including word-level language models where parts of the model (the embedding layers) are often very sparse, which involves very different communication constraints. We are also working on further improving the performance of synchronous training like combining gradients from multiple workers sharing the same machine before sending them to the parameter servers to reduce the communication overhead. An alternative of using time-outs instead of backup workers is also being explored.

## References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`.

Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Christopher M De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in Neural Information Processing Systems*, pp. 2674–2682, 2015.

J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems, NIPS*, 2012.

Jeffrey Dean and Luiz Andr Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. URL `http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext`.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

John Duchi, Michael I Jordan, and Brendan McMahan. Estimation, optimization, and parallelism when data is sparse. In *Advances in Neural Information Processing Systems*, pp. 2832–2840, 2013.

G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29:82–97, 2012.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML*, 2015.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: Asynchronous parallel saga. *arXiv preprint arXiv:1606.04809*, 2016.

Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, 2014.

Horia Mania, Xinghao Pan, Dimitris Papailiopoulos, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Perturbed iterate analysis for asynchronous stochastic optimization. *arXiv preprint arXiv:1507.06970*, 2015.

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328*, 2016.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex J Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances in Neural Information Processing Systems*, pp. 2647–2655, 2015.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. In *International Journal of Computer Vision*, 2015.

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *ArXiv 1512.00567*, 2016.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 84–97. ACM, 2016.

Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pp. 685–693, 2015a.

Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950*, 2015b.

Yuchen Zhang and Michael I Jordan. Splash: User-friendly programming interface for parallelizing stochastic algorithms. *arXiv preprint arXiv:1506.07552*, 2015.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pp. 2595–2603, 2010.

# A DETAILS OF MODELS AND TRAINING

## A.1 MNIST CNN, SECTION 2.1

The model used in our experiments is a 4-layer CNN that have 3x3 filters with max-pooling and weight normalization in every layer. We trained the model with SGD for 25 epochs and evaluated performance on the exponential moving average $\bar{\theta}$ using a decay rate of $\alpha = 0.9999$. Initial learning rate was set to be 0.1 and linearly annealed to 0 in the last 10 epochs. We also used small image rotations and zooms as a data augmentation scheme.

## A.2 INCEPTION, SECTION 3.1

For our straggler experiments, we trained the Inception (Szegedy et al., 2016) model on the ImageNet Challenge dataset (Russakovsky et al., 2015). 10 parameter servers were used, and each worker was equipped with a k40 GPU.

The underlying optimizer was RMSProp with momentum, with decay of 0.9 and momentum of 0.9. Mini-batch size $B = 32$ was used. Initial learning rates $\gamma_0$ were set at $0.045N$, which we found to provide good test precisions for Inception. Learning rates were also exponentially decreased with decay rate $\beta = 0.94$ as $\gamma_0 \beta^{tN/(2T)}$, where $T = |\mathcal{X}|/B$ is the number of mini-batches in the dataset.

Test precisions were evaluated on the exponential moving average $\bar{\theta}$ using $\alpha = 0.9999$.

## A.3 INCEPTION, SECTION 4.2

For experiments comparing Async-Opt and Sync-Opt on the Inception model in Section 4.2, each worker is equipped with a k40 GPU. For $N + b = 53$ workers, 17 parameter servers were used; for $N + b = 106$ workers, we used 27 parameter servers; and 37 parameter servers were used for $N + b = 212$.

In the asynchronous training mode, gradient clipping is also needed for stabilization, which requires each worker to collect the gradient across all layers of the deep model, compute the global norm $||G||$ and then clip all gradient accordingly. However, synchronization turns out to be very stable so gradient clipping is no longer needed, which means that we can pipeline the update of parameters in different layers: the gradient of top layers' parameters can be sent to parameter servers while concurrently computing gradients for the lower layers.

The underlying optimizer is RMSProp with momentum, with decay of 0.9 and momentum of 0.9. Mini-batch size $B = 32$ was used. Initial learning rates $\gamma_0$ for Async-Opt were set to 0.045; for Sync-Opt, we found as a rule-of-thumb that a learning rate of $0.045N$ worked well for this model. Learning rates were then exponentially decayed with decay rate $\beta = 0.94$ as $\gamma_0 \beta^{t/(2T)}$ for Async-Opt, where $T = |\mathcal{X}|/B$ is the number of mini-batches in the dataset. For Sync-Opt, we learning rates were also exponentially decreased at rate of $\gamma_0 \beta^{tN/(2T)}$, so that the learning rates after computing the same number of datapoints are comparable for Async-Opt and Sync-Opt.

Test precisions were evaluated on the exponential moving average $\bar{\theta}$ using $\alpha = 0.9999$.

## A.4 PIXELCNN, SECTION 4.3

The PixelCNN (Oord et al., 2016) model was trained on the CIFAR-10 (Krizhevsky & Hinton, 2009) dataset. Configurations of $N + b = 1, 8, 16$ workers each with a k80 GPU, and 10 parameter servers were used. For Sync-Opt, we always used $b = 1$ backup worker. The underlying optimizer is RMSProp with momentum, using decay of 0.95 and momentum of 0.9. Initial learning rates $\gamma_0$ were set to $1e - 4$ and slowly decreased to $3e - 6$ after 200,000 iterations. Mini-batch size $B = 4$ was used.