Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study

Suyog Gupta*, Wei Zhang*
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
*Equal Contribution
{suyog,weiz}@us.ibm.com

Fei Wang
Weill Cornell Medical College
New York City, NY 10065, USA
feiwang03@gmail.com

Abstract

Deep learning with a large number of parameters requires distributed training, where model accuracy and runtime are two important factors to be considered. However, there has been no systematic study of the tradeoff between these two factors during the model training process. This paper presents Rudra, a parameter server based distributed computing framework tuned for training large-scale deep neural networks. Using variants of the asynchronous stochastic gradient descent algorithm we study the impact of synchronization protocol, stale gradient updates, minibatch size, learning rates, and number of learners on runtime performance and model accuracy. We introduce a new learning rate modulation strategy to counter the effect of stale gradients and propose a new synchronization protocol that can effectively bound the staleness in gradients, improve runtime performance and achieve good model accuracy. Our empirical investigation reveals a principled approach for distributed training of neural networks: the mini-batch size per learner should be reduced as more learners are added to the system to preserve the model accuracy. We validate this approach using commonly-used image classification benchmarks: CIFAR10 and ImageNet.

1 Introduction

Deep neural network based models have achieved unparalleled accuracy in cognitive tasks such as speech recognition, object detection, and natural language processing LeCun *et al.* [2015]. Recent years have seen a resurgence of interest in deploying large-scale computing infrastructure designed specifically for training deep neural networks. Some notable efforts in this direction include distributed computing infrastructure using thousands of CPU cores Chilimbi *et al.* [2014]; Dean *et al.* [2012], high-end graphics processors (GPUs) Krizhevsky *et al.* [2012], or a combination of CPUs and GPUs Coates *et al.* [2013].

Contrary to the popular belief, among the system researchers, that asynchrony necessarily improves model accuracy, we find that adopting the approach of scale-out deep learning using asynchronous-SGD, gives rise to complex interdependencies between the training algorithm's hyperparameters and the distributed implementation's design choices (synchronization protocol, number of learners), ultimately impacting the neural network's accuracy and the overall system's runtime performance.

In this paper we present *Rudra*, a parameter server based deep learning framework to study these interdependencies and undertake an empirical evaluation with public image classification benchmarks. Our key contributions are:

1. A systematic technique (vector clock) for quantifying the staleness of gradient descent parameter updates.
2. An investigation of the impact of the interdependence of training algorithm's hyperparameters (mini-batch size, learning rate (gradient descent step size)) and distributed implementation's parameters (gradient staleness, number of learners) on the neural network's classification accuracy and training time.
3. A new learning rate tuning strategy that reduces the effect of stale parameter updates.
4. A new synchronization protocol to reduce network bandwidth overheads while achieving good classification accuracy and runtime performance.
5. An observation that to maintain a given level of model accuracy, it is necessary to reduce the mini-batch size as the number of learners is increased. This suggests a hard limit on the amount of parallelism that can be exploited in training a given model.

2 Design and Implementation

2.1 Terminology

- Parameter Server: a server that holds the model weights. The parameter server collects gradients from learners and updates the weights accordingly.
- Learner: A computing process that can calculate weight updates (gradients).
- μ: mini-batch size.
- α: learning rate.
- λ: number of learners.
- Epoch: a pass through the entire training dataset.

- Timestamp: each weight update increments the timestamp by 1. The timestamp of a gradient is the same as the timestamp of the weight used to compute the gradient.
- σ: staleness of the gradient. A gradient with timestamp ts_i is pushed to the parameter server with current weight timestamp ts_j, where $ts_j \geq ts_i$. We define the staleness of this gradient σ as $j - i$.
- $\langle\sigma\rangle$, average staleness of gradients. The timestamps of the set of n gradients that triggers the advancement of weights timestamp from ts_{i-1} to ts_i form a vector clock$\langle ts_{i_1}, ts_{i_2}, ..., ts_{i_n}\rangle$, where $max\{i_1, i_2, ..., i_n\} < i$. The average staleness of gradients $\langle\sigma\rangle$ is defined as:

$$\langle\sigma\rangle = (i-1) - mean(i_1, i_2, ..., i_n) \tag{1}$$

- Hardsync protocol: To advance weights timestamp from ts_i to ts_{i+1}, each learner calculates exactly one mini-batch and sends its gradient $\nabla\theta_l$ to the parameter server. The parameter server averages the gradients and updates the weights according to Equation (2), then broadcasts the new weights to all learners. Staleness in the hardsync protocol is always zero.

$$\begin{aligned} \nabla\theta^{(k)}(i) &= \frac{1}{\lambda}\sum_{l=1}^{\lambda}\nabla\theta_l^{(k)} \\ \theta^{(k)}(i+1) &= \theta^{(k)}(i) - \alpha\nabla\theta^{(k)}(i) \end{aligned} \tag{2}$$

- Async protocol: Each learner calculates the gradients and asynchronously pushes/pulls the gradients/weights to/from parameter server. The Async weight update rule is given by:

$$\begin{aligned} \nabla\theta^{(k)}(i) &= \nabla\theta_l^{(k)}, L_l \in L_1, ..., L_\lambda \\ \theta^{(k)}(i+1) &= \theta^{(k)}(i) - \alpha\nabla\theta^{(k)}(i) \end{aligned} \tag{3}$$

  Gradient staleness may be hard to control due to the asynchrony in the system. Dean *et al.* [2012] describe *Downpour SGD*, an implementation of the Async protocol for a commodity scale-out system in which the staleness can be as large as hundreds.

- n-softsync protocol: Each learner pulls the weights from the parameter server, calculates the gradients and pushes the gradients to the parameter server. The parameter server updates the weights after collecting at least $c = \lfloor(\lambda/n)\rfloor$ gradients. The splitting parameter n can vary from 1 to λ. The n-softsync weight update rule is given by:

$$\begin{aligned} c &= \lfloor(\lambda/n)\rfloor \\ \nabla\theta^{(k)}(i) &= \frac{1}{c}\sum_{l=1}^{c}\nabla\theta_l^{(k)}, L_j \in L_1, ..., L_\lambda \\ \theta^{(k)}(i+1) &= \theta^{(k)}(i) - \alpha\nabla\theta^{(k)}(i) \end{aligned} \tag{4}$$

  In a homogeneous cluster where each learner proceeds at roughly the same speed, the staleness of the model is expected to be n. Note that when n is equal to λ, the weight update rule at the parameter server is exactly the same as in Async protocol.

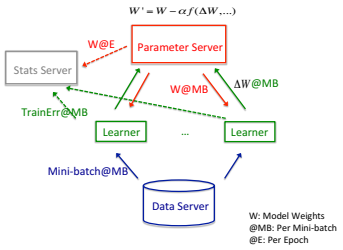
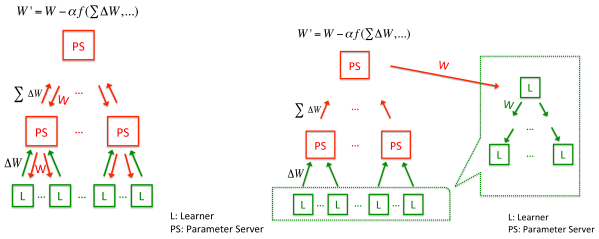


Figure 1: Rudra-base architecture



(a) Rudra-adv architecture (b) Rudra-adv* architecture

Figure 2: Rudra-adv and Rudra-adv* architecture

2.2 Rudra System Architecture

Rudra-base Figure 1 illustrates the base-line parameter server design that we use to study the interplay of hyperparameter tuning and system scale-out factor. *Parameter Server* is a multithreaded process, that accumulates gradients from each learner and applies update rules according to Equations (2–4) (i.e., hardsync and n-softsync protocols). *Learner* is a single-process multithreaded SGD solver, it calculates the gradients and sends the gradients along with their timestamp to the parameter server. Learner retrieves weights from the Parameter Server when newer weights are generated. *Data Server* hosts training and testing data. *Statistics Server* records training statistics (e.g., training error).

Rudra-adv To alleviate the network traffic to parameter server, we build a parameter server group that forms a tree. We co-locate each tree leaf node on the same node as the learners for which it is responsible. Each node in the parameter server group is responsible for averaging the gradients sent from its learners and relaying the averaged gradient to its parent. The root node in the parameter server group is responsible for applying weight update and broadcasting the updated weights. Each non-leaf node pulls the weights from its parent and responds to its children's weight pulling requests. Figure 2(b) illustrates the system architecture for Rudra-adv.

Rudra-adv*. We built Rudra-adv* to further improve the runtime performance of Rudra-adv in two ways: (1) Broadcast weights within learners to utilize network links between learners. (2) Asynchronous `pushGradient` and `pullWeights`. Figure 2(b) illustrates the system architecture for Rudra-adv*. Different from Rudra-adv, each learner continuously receives weights from the weights downcast tree, which consists of the top level parameter server node and all the learners.

3 Methodology

3.1 Hardware and Software Environment

We deploy the Rudra distributed deep learning framework on a P775 supercomputer. Each node of this system contains four eight-core 3.84 GHz POWER7 processors, one optical connect controller chip and 128 GB of memory. A single node has a theoretical floating point peak performance of 982 Gflop/s, memory bandwidth of 512 GB/s and bi-directional interconnect bandwidth of 192 GB/s.The cluster operating system is Red Hat Enterprise Linux 6.4. To compile and run Rudra we used the IBM xlC compiler version 12.1 with the `-O3 -q64 -qsmp` options, ESSL for BLAS subroutines, and IBM MPI (IBM Parallel Operating Environment 1.2).

3.2 Benchmark Datasets and Neural Network Architectures

To evaluate Rudra's scale-out performance we employ two commonly used image classification benchmark datasets: `CIFAR10` Krizhevsky and Hinton [2009] and `ImageNet` et al [2015]. The neural network architecture that we used for `CIFAR10` closely mimics the `CIFAR10` model (cifar10_full.prototxt) available as a part of the open-source Caffe deep learning package Jia *et al.* [2014]. The neural network is trained using momentum-accelerated mini-batch SGD with a batch size of 128 and momentum set to 0.9. The base learning rate α_0 is set to 0.001 are reduced by a factor of 10 after the 120^{th} and 130^{th} epoch. For `ImageNet`, we use the neural network architecture introduced in Krizhevsky *et al.* [2012] consisting of 5 convolutional layers and 3 fully-connected layers. The initial learning rate α_0 is set equal to 0.01 and reduced by a factor of 10 after the 15^{th} and 25^{th} epoch.

4 Evaluation

In this section we present results of evaluation of our scale-out deep learning training implementation. For an initial design space exploration, we use the `CIFAR10` dataset and Rudra-base system architecture (Section 4.1 and Section 4.2). Subsequently we extend our findings to the `ImageNet` dataset using the Rudra-adv and Rudra-adv* system architectures (Section 4.3).

4.1 Staleness-Aware Learning Rate Tuning

In our experiments with the n-softsync protocol we found it beneficial, and at times necessary, to modulate the learning rate α to take into account the staleness of the gradients. For the n-softsync protocol, we set the learning rate as:

$$\alpha = \alpha_0/\langle\sigma\rangle = \alpha_0/n \tag{5}$$

where α_0 is the learning rate used for the baseline (control) experiment: $\mu = 128$, $\lambda = 1$. Figure 3 shows a set of representative results illustrating the benefits of adopting this learning rate modulation strategy. We show the evolution of the test error on the `CIFAR10` dataset as a function of the training epoch for two different configurations of the n-softsync protocol ($n = 4$, $n = 30$) and set the number of learners, $\lambda = 30$. In both these configurations, setting the learning rate in accordance with equation (5) results in lower test error as compared with the cases where the learning rate is set to α_0. Surprisingly, the configuration 30-softsync, $\lambda = 30$, $\alpha = \alpha_0$ fails to converge and shows a constant high error rate of 90%. Reducing the learning rate by a factor $\langle\sigma\rangle = n = 30$ makes the model with much lower test error.

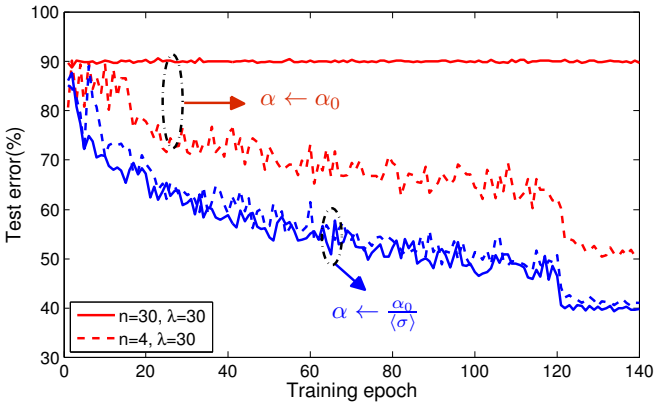


Figure 3: *Effect of learning rate modulation strategy*: Dividing the learning rate by the average staleness aids in better convergence and achieves lower test error when using the n-softsync protocol. Number of learners, $\lambda = 30$; mini-batch size $\mu = 128$.

4.2 $\mu\lambda$ = constant

It is trivial to conclude that training with different configurations, in which $\mu\lambda$ = constant, yields similar (if not the same) model accuracy for the hardsync protocol.

Interestingly, we find that even for the n-softsync protocol, configurations that maintain $\mu\lambda$ = constant achieve comparable test errors. In Table 2 we report the test error at the end of 140 epochs for configurations with $\mu\lambda$ = constant. For instance, Table 2 shows that when $\mu\lambda \approx 128$, the test error stays $\sim$18-19%. Table 2 also shows that the test error increases monotonically with the $\mu\lambda$ product. These results reveal the scalability limits under the constraints of preserving the model accuracy. These results help define a principled approach for distributed training of neural networks: *the mini-batch size per learner should be reduced as more learners are added to the system in way that keeps $\mu\lambda$ product constant. In addition, the learning rate should be modulated to account for stale gradients.* From machine learning perspective, this points to an interesting research direction on designing optimization algorithm and learning strategies that perform well with large mini-batch sizes.

4.3 Results on `ImageNet` Benchmark

The computational cost of training `ImageNet` prohibits an exhaustive state space exploration of the interplay between model accuracy and system runtime performance. The baseline configuration ($\mu = 256$, $\lambda = 1$) takes 54 hours/epoch. Guided by the results of section 4.2, we first consider a configuration with $\mu = 16$, $\lambda = 18$ and employ the Rudra-base architecture with hardsync protocol (*base-hardsync*). This configuration performs training at the speed of $\sim$330 minutes/epoch and achieves a top-5 error of 20.85%, matching the accuracy of the baseline configuration ($\mu = 256$, $\lambda = 1$).

Configuration	Architecture	μ	λ	Synchronization protocol	Validation error(top-1)	Validation error (top-5)	Training time (minutes/epoch)
base-hardsync	Rudra-base	16	18	Hardsync	44.35%	20.85%	330
base-softsync	Rudra-base	16	18	1-softsync	45.63%	22.08%	270
adv-softsync	Rudra-adv	4	54	1-softsync	46.09%	22.44%	212
adv**-softsync*	Rudra-adv*	4	54	1-softsync	46.53%	23.38%	125

Table 1: *Results on* `ImageNet` *benchmark*: Validation error at the end of 30 epochs and training time per epoch for different configurations.

	σ	μ	λ	Test error	Training time(s)
$\mu\lambda \approx 128$	**1**	**4**	**30**	**18.09%**	**1573**
	30	4	30	18.41%	2073
	18	8	18	18.92%	2488
	10	16	10	18.79%	3396
	4	32	4	18.82%	7776
	2	64	2	17.96%	13449
$\mu\lambda \approx 256$	**1**	**8**	**30**	**20.04%**	**1478**
	30	8	30	19.65%	1509
	18	16	18	20.33%	2938
	10	32	10	20.82%	3518
	4	64	4	20.70%	6631
	2	128	2	19.52%	11797
	1	128	2	19.59%	11924
$\mu\lambda \approx 512$	**1**	**16**	**30**	**23.25%**	**1469**
	30	16	30	22.14%	1502
	18	32	18	23.63%	2255
	10	64	10	24.08%	2683
	4	128	4	23.01%	7089
$\mu\lambda \approx 1024$	**1**	**32**	**30**	**27.16%**	**1299**
	30	32	30	27.27%	1420
	18	64	18	28.31%	1713
	1	128	10	29.83%	2551
	10	128	10	29.90%	2626

Table 2: *Results on* `CIFAR10` *benchmark*: Test error at the end of 140 epochs and training time for (σ, μ, λ) configurations with $\mu\lambda =$ constant.

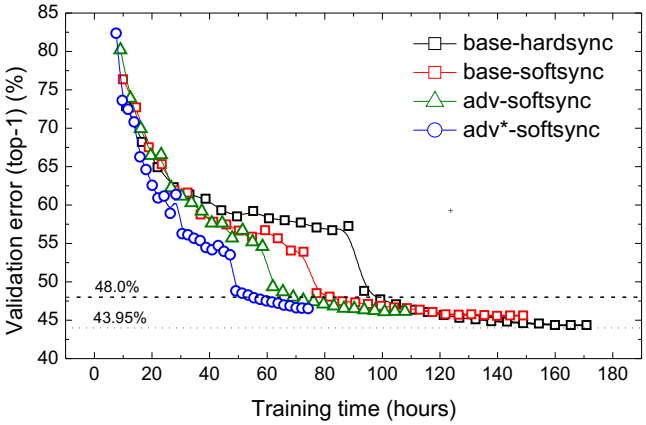


Figure 4: *Results on* `ImageNet` *benchmark:* Error on the validation set as a function of training time for the configurations listed in Table 1

Training using the 1-softsync protocol with mini-batch size of $\mu = 16$ and 18 learners takes $\sim$270 minutes/epoch, reaching a top-1 (top-5) accuracy of 45.63% (22.08%) by the end of 30 epochs (*base-softsync*).

$\lambda = 54$ learners, each processing a mini-batch size $\mu = 4$, train at $\sim$212 minutes/epoch when using Rudra-adv architecture and 1-softsync protocol (*adv-softsync*). As in the case of Rudra-base, the average staleness in the gradients is close to 1 and this configuration also achieves a top-1(top-5) error of 46.09% (22.44%). The Rudra-adv* improves the runtime performance to $\sim$125 minutes/epoch, and its top-1 validation error is (46.53%).

Table 1 summarizes the results obtained for the 4 configurations discussed above. Figure 4 compares the evolution of the top-1 validation error during training for the 4 different configuration summarized in Table 1. The training speed follows the order *adv***-softsync* $>$ *adv-softsync* $>$ *base-softsync* $>$ *base-hardsync*. As a result, *adv***-softsync* is the first configuration to hit the 48% validation error mark. Configurations other than *base-hardsync* show marginally higher validation error compared with the baseline.

5 Conclusion

In this paper, we empirically studied the interplay of hyper-parameter tuning and scale-out in three protocols for communicating model weights in asynchronous stochastic gradient descent. We divide the learning rate by the average staleness of gradients, resulting in faster convergence and lower test error. Our experiments show that the 1-*softsync* protocol (in which the parameter server accumulates λ gradients before updating the weights) minimizes gradient staleness and achieves the lowest runtime for a given test error. We found that to maintain a model accuracy, it is necessary to reduce the mini-batch size as the number of learners is increased. This suggests an upper limit on the level of parallelism that can be exploited for a given model, and consequently a need for algorithms that permit training over larger batch sizes. We provide theoretical justification of these empirical results in a separate paper Zhang *et al.* [2017].

References

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. OSDI'14, pages 571–582, 2014.

Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th ICML*, pages 1337–1345, 2013.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, MarcAurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

Olga Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, pages 1–42, 2015.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Wei Zhang, Suyog Gupta, Xiangru Lin, Ji Liu, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study and its theoretical underpinning. *Knowledge and Information Systems*, (In submission), 2017.