Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning

Arissa Wongpanich
arissa@berkeley.edu
Computer Science Division
UC Berkeley
Berkeley, CA, USA

Yang You
youyang@cs.berkeley.edu
Computer Science Division
UC Berkeley
Berkeley, CA, USA

James Demmel
demmel@cs.berkeley.edu
Computer Science Division
UC Berkeley
Berkeley, CA, USA

ABSTRACT

In recent years, the field of machine learning has seen significant advances as data becomes more abundant and deep learning models become larger and more complex. However, these improvements in accuracy [2] have come at the cost of longer training time. As a result, state-of-the-art models like OpenAI's GPT-2 [18] or AlphaZero [20] require the use of distributed systems or clusters in order to speed up training. Currently, there exist both asynchronous and synchronous solvers for distributed training. In this paper, we implement state-of-the-art asynchronous and synchronous solvers, then conduct a comparison between them to help readers pick the most appropriate solver for their own applications. We address three main challenges: (1) implementing asynchronous solvers that can outperform six common algorithm variants, (2) achieving state-of-the-art distributed performance for various applications with different computational patterns, and (3) maintaining accuracy for large-batch asynchronous training. For asynchronous algorithms, we implement an algorithm called EA-wild, which combines the idea of non-locking wild updates from Hogwild! [19] with EASGD. Our implementation is able to scale to 217,600 cores and finish 90 epochs of training the ResNet-50 model on ImageNet in 15 minutes (the baseline takes 29 hours on eight NVIDIA P100 GPUs). We conclude that more complex models (e.g., ResNet-50) favor synchronous methods, while our asynchronous solver outperforms the synchronous solver for models with a low computation-communication ratio. The results are documented in this paper; for more results, readers can refer to our supplemental website [1].

ACM Reference Format:
Arissa Wongpanich, Yang You, and James Demmel. 2020. Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020), January 15–17, 2020, Fukuoka, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3368474.3368498

[1] www.sites.google.com/view/rethinking-async-solvers

HPCAsia2020, January 15–17, 2020, Fukuoka, Japan

ACM ISBN 978-1-4503-7236-7/20/01…$15.00
https://doi.org/10.1145/3368474.3368498

1 INTRODUCTION

In recent years, the field of machine learning has seen significant advances as data becomes more abundant and deep learning models become larger and more complex. However, these improvements in accuracy [2] have come at the cost of longer training time. As a result, state-of-the-art models like OpenAI's GPT-2 [18] or AlphaZero [20] require the use of distributed systems or clusters in order to speed up training. These systems often consist of hardware accelerators (such as GPUs or TPUs) which have limited individual on-chip memory and must fetch data from either CPU memory or the disk of the server. In such systems, the limited computing performance of one server quickly becomes a bottleneck for large datasets such as ImageNet. Therefore, distributed systems commonly use tens or hundreds of host servers connected by high-speed interconnects, each with multiple accelerators. In this paper, we conduct a study to determine the trade-offs between different training methods on these large-scale systems. Currently, there exist both asynchronous and synchronous solvers for distributed training. We first implement the best existing asynchronous and synchronous solvers, then conduct a comparison between them to help readers select the most appropriate solver for their own applications. We address three main challenges in this paper. Firstly, the implementation of existing state-of-the-art asynchronous solvers poses a challenge because standard frameworks like TensorFlow only support Downpour SGD or parameter server [4]. There are several variants of these asynchronous solvers (e.g. Hogwild!, Async Momentum, EASGD) and each one claims to achieve state-of-the-art performance. Secondly, achieving state-of-the-art distributed performance for all the applications poses another challenge because the computation-communication ratio varies for different applications. The computation-communication ratio is a measure of the model's complexity compared with its size and is used to assess the effectiveness of each solver on various applications. Here, computation refers to floating point operations, and communication refers to the data movement between different levels of memory or between different host servers over the networks in a cluster. Lastly, in order to avoid accuracy loss when training with extremely large batch sizes, we must implement large-batch asynchronous training algorithms and design an auto-tuning scheduler for hyper-parameters [10].

For the asynchronous algorithms, we first study Elastic Averaging Stochastic Gradient Descent (EASGD) [27] since its scalability can potentially be improved. In the original EASGD, communication is carried out with a round-robin method, which is inefficient on high-performance clusters.

To overcome this communication overhead, we use the idea of non-locking Hogwild! updates [19] with EASGD to implement an EA-wild algorithm. The non-locking updates significantly boost the throughput of the parallel system, and our implementation is able to scale up the deep neural network training to 217,600 cores and finish 90 epochs of ImageNet training with the ResNet-50 model in 15 minutes (the baseline takes 29 hours on eight NVIDIA P100 GPUs).

We conclude that more complex models (e.g., ImageNet data with ResNet-50) favor the Sync solver. The Async solver outperforms the Sync solver for less complex models with a low computation-communication ratio.

The results are documented in this paper; for more results, readers can refer to our supplemental website [2].

Notation. Throughout this paper, we use P to denote the number of machines/processors, $\boldsymbol{w}$ to denote the parameters (weights of the models), $\boldsymbol{w}^j$ to denote the local parameters on j-th worker, and $\tilde{\boldsymbol{w}}$ to denote the global parameters. We use $\Delta\boldsymbol{w}^j$ to denote the stochastic gradient evaluated at the j-th worker.

2 BACKGROUND AND RELATED WORK

2.1 Parallelization of Deep Neural Networks

There are two major directions for parallelizing DNN: data parallelism and model parallelism.

Data Parallelism. Data parallelism stores data across machines. The data is split into P different parts, and each part is stored on a different machine. A local copy of the weights ($\boldsymbol{w}^j$) is also stored on each machine.

Model Parallelism With model parallelism, the neural network is partitioned into P pieces and distributed across each machine. This differs from data parallelism, where each machine contains a copy of the neural network.

For deep-narrow neural networks, the main form of parallelism comes from data parallelism. For wide-shallow networks, the main form of parallelism comes from model parallelism. Since most current neural networks are deep-narrow, for the purposes of this paper we focus on data parallelism.

2.2 Recent Hardware for Distributed Learning

Researchers commonly train their models on a server with multiple GPUs, such as NVIDIA's P100 and V100 GPUs; on the other hand, recently introduced many-core chips like Google TPU and Intel Knights Landing (KNL) also offer good performance. Researchers can use a cluster with hundreds of thousands of these chips (e.g. CPU, GPU, KNL, TPU) to train deep neural networks. To the best of our knowledge, almost all the state-of-the-art models since 2018 (e.g. Google Bert, OpenAI GPT-2, AlphaZero) have been trained by clusters. To make full use of these massive parallel hardware resources, efficient parallel and distributed solvers are necessary.

2.3 State-of-the-Art Asynchronous Solvers

Since the first large-scale asynchronous solver was implemented by Google Brain [4], several different asynchronous solvers have been used in industry. To our best knowledge, the most widely-used methods are Parameter-Servers, Hogwild! SGD and EASGD.

2.3.1 Hogwild! SGD. Classical Stochastic Gradient Descent (SGD) uses a lock between different weight updates to avoid thread conflicts [12]. As datasets become larger and the number of threads increases, this locking scheme often becomes a bottleneck that prevents SGD from using a large number of threads. In Hogwild! SGD [19], this lock is removed and all the threads can update the global parameters asynchronously at the same time. As a result, the algorithm has a much faster updating speed. Although Hogwild! SGD updates have conflicting parameter accesses, it has been proved [19] that the algorithm still converges to the optimum under certain assumptions.

2.3.2 Elastic Averaging SGD (EASGD). The EASGD method [27] has been proposed as a variant of SGD for distributed systems. EASGD formulates the problem as

$$\min_{\boldsymbol{w}^1,\ldots,\boldsymbol{w}^P,\tilde{\boldsymbol{w}}} \sum_{j=1}^{P}(f_j(\boldsymbol{w}^j) + \frac{\rho}{2}\|\boldsymbol{w}^j - \tilde{\boldsymbol{w}}\|_2^2), \tag{1}$$

where each local function associated with the local data is $f_j(\cdot)$. The benefit of this reformulation is that each local function $f_j(\cdot)$ is only related to the local model $\boldsymbol{w}^j$. To solve (1), Zhang et al. [27] has proposed the following scheme. Local workers conduct the local SGD updates with respect to $\boldsymbol{w}^j$:

$$\boldsymbol{w}_{t+1}^j = \boldsymbol{w}_t^j - \eta(\nabla f_j(\boldsymbol{w}_t^j) + \rho(\boldsymbol{w}_t^j - \tilde{\boldsymbol{w}}_t)), \tag{2}$$

which will only use local data f_j.

To update the global parameters $\tilde{\boldsymbol{w}}$ (which requires communication between the workers and master), the scheme proposed by Zhang et al. uses a round-robin strategy for scheduling the updates, i.e. the update of worker j to the master cannot be started until the update of worker $j-1$ to the master is finished [27].

Each update to the global parameter can be written as

$$\tilde{\boldsymbol{w}}_{t+1} = \tilde{\boldsymbol{w}}_t + \eta\rho(\boldsymbol{w}_t^j - \tilde{\boldsymbol{w}}_t). \tag{3}$$

The ρ in Equation (2) and Equation (3) is a term that connects global and local parameters. As a result, EASGD allows the local workers to encourage exploration (small ρ) while the master may carry out more exploitation.

3 IMPLEMENTING ASYNCHRONOUS AND SYNCHRONOUS SOLVERS

3.1 EA-wild

In the original EASGD algorithm, although the updates follow a round-robin scheme, it is still possible that $\boldsymbol{w}_t^j$ (the parameters of the j-th worker) arrives at the time when the master is still performing the update of an earlier-arrived worker. To avoid conflicting updates, EASGD uses a lock to ensure that the update of $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} + \eta\rho(\boldsymbol{w}^j - \tilde{\boldsymbol{w}})$ must be finished before conducting another update $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} + \eta\rho(\boldsymbol{w}^i - \tilde{\boldsymbol{w}})$. In a multi-GPU system, this means that each GPU must wait until the previous GPU finishes the update to the global parameter in memory. In a distributed system, this means that when machine-$i+1$ wants to update the local model to master, it must wait until machine-i finishes its communication and updating. As

[2] www.sites.google.com/view/rethinking-async-solvers

a result, the lock for parameter updates will slow down the overall system, which is confirmed in our experiments.

To solve this problem in both multi-GPU and distributed systems, we implement an EA-wild algorithm by removing the lock for the parameter updates $\tilde{\boldsymbol{w}} \leftarrow \tilde{\boldsymbol{w}} + \eta\rho(\boldsymbol{w}^j - \tilde{\boldsymbol{w}})$. The framework of the EA-wild method is shown in Algorithm 1. Note that we drop all the iteration numbers t here since the updates to the master are carried out in an asynchronous manner without any locking. After removing the lock, our algorithm has much a higher throughput since there is no waiting time for each global parameter update. However, removing the lock also results in harder theoretical analysis, since there may be conflict between updates from multiple devices. We also partition the workers into different groups. Within each group, the workers communicate synchronously with each other. We then pick one worker from each group to push the EA-wild updating to the global parameter server. The grouping method can be predetermined or random. It is worth noting that EA-wild is a variant of Hogwild! EASGD [23]. We additionally provide a strong theoretical guarantee to EA-wild (Section 3.3).

Algorithm 1 EA-wild

Input: Samples and labels: $\{X_i, y_i\}$ $i \in 1, ..., n$, batch size: B, number of workers: P.
Output: Model weight $\boldsymbol{w}$
Initialize $\tilde{\boldsymbol{w}}_1 = \boldsymbol{w}_1^1, \ldots, \boldsymbol{w}_1^P$ on Master and Workers
for $t = 1, 2, \ldots$ **do**
Execute by Worker j**:**
Picks B samples with equal probability
Master sends $\tilde{\boldsymbol{w}}$ to Worker
Master gets $\boldsymbol{w}^j$ from Worker
Workers compute gradient $\Delta\boldsymbol{w}^j$ on selected samples
Workers update $\boldsymbol{w}^j = \boldsymbol{w}^j - \eta\rho(\boldsymbol{w}^j - \tilde{\boldsymbol{w}}) - \eta\Delta\boldsymbol{w}^j$
Execute by master:
Master gets $\boldsymbol{w}^j$ from j-th worker
Master updates $\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{w}} + \eta\rho(\boldsymbol{w}^j - \tilde{\boldsymbol{w}})$ **without lock**
end for

3.2 Sync Solver: efficient all-reduce operation

Assume we have P machines, and we use the standard method to implement Sync SGD, using the same computational pattern as other common optimizers such as momentum, AdaGrad, or Adam. Each machine in the cluster has a copy of weights $\boldsymbol{w}$ and B/P data samples where B is the global batch size. Each machine computes its local gradients $\Delta\boldsymbol{w}^j$ at each iteration. The algorithm must get the sum of all local gradients and broadcast this sum to all the machines. Then each machine updates the local weights by $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta/P \sum_{j=1}^{P} \Delta\boldsymbol{w}^j$. The sum of gradients can be implemented as an all-reduce operation. The ring all-reduce implementation which we decided to use performs poorly when we increase the number of machines beyond 1K. We used a hierarchical approach that divides P machines into P/G groups (we use G=4, 8, or 16 depending on the data size). There are three steps in our implementation: intra-group reduction, inter-group all-reduce, and intra-group broadcast.

3.3 Convergence Rate of EA-wild

For our discussion of the convergence rate of the EA-wild algorithm, we only consider convex functions. For nonconvex functions such as deep neural networks, it is generally hard to guarantee the convergence to the global optimum, and thus we leave this as future work.

As our EA-wild algorithm also utilizes the lock-free approach introduced by the Hogwild! algorithm [19], we refer the reader to the proof in the original paper [17]. Like the Hogwild! algorithm, our EA-wild algorithm also achieves nearly linear speedup in the number of processors since it is a lock-free approach. Note that the functions covered by our theory are more general than the original proof in the EASGD paper [27].

We restate the convergence bounds derived in the Hogwild! paper [17], based on the following finite sum problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \quad f(\boldsymbol{w}) := \textstyle\sum_{e\in E} f_e(\boldsymbol{w}_e), \tag{4}$$

where $\boldsymbol{w}$ is the weight, and each e is a small subset of $\{1, \ldots, d\}$. For example, in linear empirical risk minimization, $f_e(\boldsymbol{w}_e) := \ell_i(\boldsymbol{w}^T\boldsymbol{x}_i)$ for a training sample $\boldsymbol{x}_i$, where e refers to the nonzero elements in $\boldsymbol{x}_i$. The function of equation (4) can be described with a hypergraph $G = (V, E)$ whose nodes are the individual components of $\boldsymbol{w}$, and each subvector $\boldsymbol{w}_e$ is an edge in graph G consisting of some subset of nodes. We also define the following notations for hypergraph G:

$$\Omega := \max_{e\in E}|e|, \tag{5}$$

$$\Delta := \frac{\max_{1\le v\le n}|\{e \in E : v \in e\}|}{|E|}, \tag{6}$$

$$\rho := \frac{\max_{e\in E}|\{\hat{e} \in E : \hat{e} \cap e \neq \phi\}|}{|E|}. \tag{7}$$

Here Ω is the size of the hyper edges, ρ denotes the maximum fraction of edges that intersect any given edge (measuring the sparsity of the graph), and Δ determines the maximum fraction of edges that intersect any variable (measuring the node regularity).

Using these defined values, the convergence bound can be summarized with the following proposition, which is the same as that found in Hogwild! [17].

Proposition 1. *Suppose in the EA-wild algorithm that the lag between when a local weight $\boldsymbol{w}_t^j$ is received by the master and when the step is used must always be less than or equal to τ. Let us define γ as*

$$\gamma = \frac{\theta\epsilon c}{2LM^2(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{\frac{1}{2}})} \tag{8}$$

for some $\epsilon > 0$ and $\theta \in (0, 1)$. Furthermore, let $D_0 := \|\boldsymbol{w}_0 - \boldsymbol{w}_*\|_2^2$ and k be an integer satisfying

$$k \ge 2LM^2(1 + 6\rho\tau + 6\tau^2\Omega\Delta^{\frac{1}{2}})\frac{\log(LD_0/\epsilon)}{c^2\theta\epsilon}. \tag{9}$$

Then after k updates of w, we have $E[f(\boldsymbol{w}_k) - f_*] \le \epsilon$.

For EA-wild, we assume $\gamma c < 1$ because even the original EASGD diverges when $\gamma c \ge 1$. In the case that $\tau = 0$, the algorithm becomes EASGD with only one worker. As with [17], we can achieve a similar rate if $\tau = o(d^{1/4})$ because ρ and Δ usually are $o(1/d)$ for sparse data. Since P workers can finish the same number of iterations P

times faster than a single worker, we have linear speedup when $P = o(d^{1/4})$. In deep neural networks, d is usually very large (on the order of more than tens of millions), so this theory suggests that we can parallelize on a large number of machines with near-linear speed up.

4 ASYNC-SYNC SOLVERS COMPARISON

In this section, we conduct experimental comparisons to demonstrate that our proposed EA-wild algorithm outperforms the original EASGD algorithm and other asynchronous solvers on both distributed systems and multi-GPU servers.

On the other hand, the results in Section 4.5 prove that our synchronous implementation also achieves state-of-the-art performance among synchronous solvers. Based on these efficient implementations, we conduct a fair comparison study of the Async solver versus the Sync solver.

4.1 Experimental Settings

Accuracy in this paper refers to Top-1 test accuracy. Time refers to wall-clock training time. One epoch refers to the algorithm statistically touching all the training samples once. For the GPU implementation, we use the Nvidia Collective Communications Library (NCCL) and Message Passing Interface (MPI) for communication. We base our KNL (Intel Knights Landing, an advanced many-core CPU) implementation on Caffe for single-machine processing and MPI for the communication among different machines on the KNL cluster. Both GPU clusters and KNL clusters use Infiniband as the interconnect network. In most situations, there is no difference between KNL programming and regular CPU programming. However, to achieve high performance for non-trivial applications, we wrote some low-level code to customize the matrix multiply operations for different layers and modified the SIMD vectorization by using the flexible vector width. We also significantly tuned the code based on architecture parameters (e.g. cache size).

The datasets we used in this paper include the MNIST [13] dataset, CIFAR-10 [11], and ImageNet [5]. The MNIST dataset is processed by the LeNet model [13] and a pure-LSTM model. The CIFAR-10 dataset is processed by the DenseNet model [8]. The ImageNet dataset is processed by the GoogLeNet model [21] and the ResNet-50 model [7].

4.2 EA-wild versus other Async Solvers

First, we want to ensure that we have a strong Async baseline. We demonstrate that our implemented EA-wild algorithm is faster than other asynchronous solvers. To conduct a fair comparison, we implement them on the same 4-GPU machine. Moreover, we make sure the different methods are nearly identical to each other, with the only difference being the schedule for how the gradients are communicated and how the weights are updated. The asynchronous solvers we wish to compare include:

- EA-wild: the EASGD with lock-free parameter updating rule.
- Hogwild!: the method proposed in [19].
- Parameter Server: Traditional Async SGD.
- Async MSGD: asynchronous SGD with momentum.
- EA-MSGD: Async EASGD with momentum.
- EASGD: the EASGD with a round-robin rule.

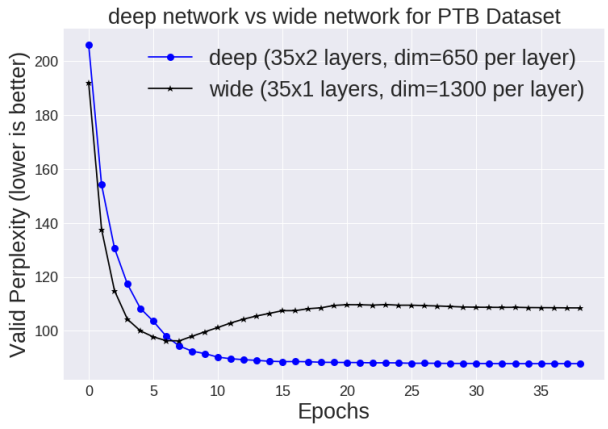

Figure 1: Given the same number of parameters, we observe that deep neural networks constantly beat wide neural networks. We use this information to maximize the parallelism in distributed training.

In our experiments comparing these different asynchronous methods against each other, our EA-wild implementation is much faster than other asynchronous solvers. However, it is worth noting that EA-wild does not beat the synchronous solver for all applications, notably those that require more complex models. In the following sections, we will describe how we ensured that the best asynchronous solver was used to conduct a fair comparison to the synchronous solver.

4.3 Exploring the maximum parallelism

Let us define two neural networks, M_1 and M_2, which both have the same number of parameters. We will assume that M_1 has more layers, and that the layers of M_2 are wider than M_1 on average. In our experiments, we found that M_1 constantly has a higher accuracy than M_2. All comparisons use the same data, hardware, and training budgets. An example is shown in Figure 1. We come to the conclusion that, given a fixed number of parameters, deeper models outperform wider models. Wide neural networks lend themselves easily to model parallelism because they create larger matrices or tensors per layer. Due to the dependency between different layers, we can parallelize the forward/backward propagation between different layers. For deep-narrow neural networks, the main parallelism comes from data parallelism. Therefore, we must maximize the global batch size for both asynchronous solvers and synchronous solvers.

4.4 Scaling to hundreds of thousands of cores

The scalability of our implementation on KNL systems using the EA-wild asynchronous algorithm is shown in Figure 2. It is clear that the training speed increases as we increase the number of cores in the KNL system. We successfully scaled the algorithm to 217,600 CPU cores (each KNL has 68 cores) and finished the 90-epoch ImageNet training with the ResNet-50 model in 15 minutes.

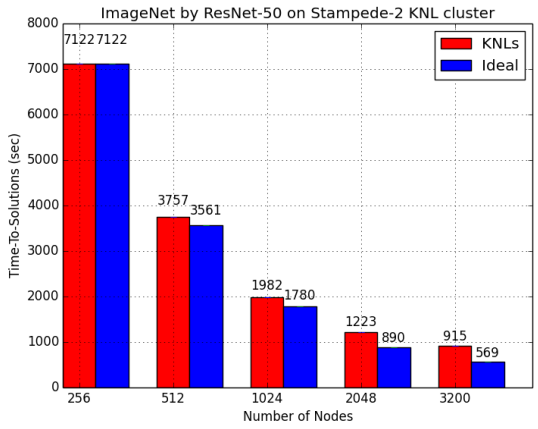

Figure 2: We scaled the DNN training to 217,600 cores (each KNL has 72 CPU cores) and finished the 90-epoch ImageNet training with ResNet-50 model in 15 minutes using the EA-wild asynchronous algorithm. Ideal training time is computed by simply dividing the total training time by the number of workers.

4.5 Async vs Sync SGD for large-batch training

For state-of-the-art deep neural network training, some researchers use asynchronous methods [4] while other researchers prefer synchronous methods [6]. Since large-batch methods can improve performance, they have recently been actively studied [1], [6], [14], [22], [26]. However, all of the existing large-batch methods use a synchronous approach. Moreover, their studies are based on limited applications (i.e. ImageNet training with ResNet-50). In this section, we conduct a comprehensive study on the comparison between asynchronous methods and synchronous methods for large-batch training. To make sure our synchronous and asynchronous implementations are correct, we use open-source frameworks such as Intel Caffe and Tensorflow as the baselines. We make sure that we achieve the results with the standard frameworks. The details of our experimental models and datasets are shown in Table 1.

4.5.1 Maintaining Accuracy. For a batch size beyond 2K, a straightforward implementation without careful learning rate scheduling often leads to accuracy loss or divergence in async solvers [10]. To minimize the accuracy loss of async solvers, we design a two-stage learning rate scheduling (Figure 3).

4.5.2 cifar10-full model for CIFAR-10 Dataset. cifar10-full is implemented by the Caffe team for fast CIFAR-10 training. In this experiment, the weight decay is 0.004, the momentum is 0.9, and we use a polynomial policy to decay the learning rate (power = 1.0). As suggested by earlier work [6], we keep all the above settings constant and only change the learning rate when we scale the batch size. The original implementation achieves 82% accuracy[3]. By using a warmup scheme and LARS [25], we are able to scale the batch size to 5K with a reasonable accuracy (Table 2). However,

[3] github.com/BVLC/caffe/tree/master/examples/cifar10

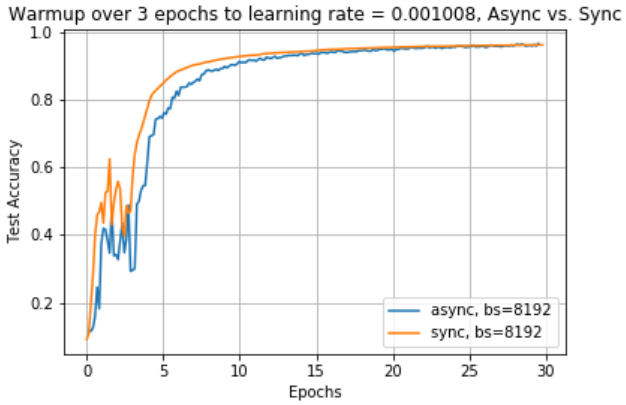

Figure 3: We use Gradient Descent Optimizer during the warmup and then Adam Optimizer once warmup is complete. We chose to use Gradient Descent Optimizer during the warmup because we can easily set the learning rate, whereas Adam Optimizer can dynamically change the learning rate based on the online gradients information. The dataset is MNIST and the model is LeNet. bs denotes batch size. Warmup means we gradually increase the batch size in the first few epochs.

Table 1: Datasets and models used in Async EA-wild method and Sync method comparison study. The accuracy shown is computed using the baseline current state-of-the-art solvers, run on a single node.

Dataset	Model	Epochs	Baseline Accuracy
CIFAR-10	cifar10-full	140	82%
MNIST	LSTM	50	98.7%
CIFAR-10	DenseNet	290	93%
ImageNet	ResNet-50	90	75.3%
ImageNet	GoogLeNet	60	68.7%

the accuracy is lower than 80% when we scale the batch size to 8K. After a comprehensive tuning of the learning rate and warmup, we observe that the Sync method's accuracy is slightly lower than the EA-wild asynchronous method for batch size = 8K (Figure 4). The Sync method uses eight machines. The Async EA-wild method uses one server and eight workers (nine machines). In this example, the Async EA-wild method slightly beats the Sync method with regards to accuracy. However, with regards to system speed, the Sync method on 8 KNLs is 1.44× faster than the Async EA-wild method on 9 KNLs for running the same 140 epochs.

4.5.3 LSTM model for MNIST Dataset. Each sample is a 28-by-28 handwritten digit image. We use a pure-LSTM model to process this dataset. We partition each image as 28-step input vectors. The dimension of each input vector is 28-by-1. Then we have a 128-by-28 transform layer before the LSTM layer, which means the actual LSTM input vector is 128-by-1. The hidden dimension of the LSTM layer is 128. The baseline achieves a 98.7% accuracy for batch size = 256. When we scale the global batch size to 8K with four machines,

Table 2: Accuracy for cifar10-full model. 140 epochs total, weight decay is 0.004, and momentum is 0.9. The Async EA-wild method beats the Sync method for batch size = 8192. A small batch can get a much better accuracy. But large-batch training is a sharp minimal problem [1]. 8192 is a huge batch size for CIFAR because it only has 50K samples.

Batch Size	Method	LR	warmup	Accuracy
100	Sync	0.001	0 epoch	82.08%
1K	Sync	0.010	0 epoch	82.12%
2K	Sync	0.081	7 epochs	82.15%
5K	Sync	0.218	17 epochs	81.15%
8K	Sync	0.450	6 epochs	74.92%
8K	Async EA-wild	0.420	6 epochs	75.51%

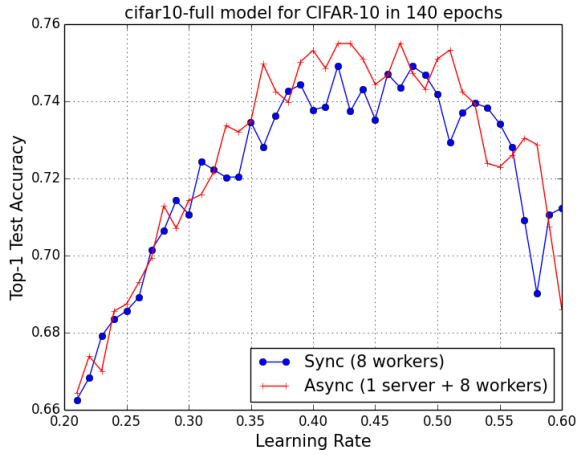

Figure 4: Accuracy for cifar10-full model. 140 epochs total, weight decay is 0.004, momentum is 0.9, and batch size is 8192. The Async EA-wild method beats the Sync method with respect to accuracy.

the Sync solver achieves 98.6% accuracy. The Async EA-wild solver with a server and four workers only achieves 96.5% accuracy for the global size of 8K. For this application, the Sync method is much better and more stable than the Async EA-wild method (Figures 5-7). As we increase the number of workers, the performance of async solvers degrades.

4.5.4 DenseNet model for CIFAR-10 dataset. We use a 40-layer DenseNet for accurate CIFAR-10 training. In this experiment, the weight decay is 0.0001, the momentum is 0.9, and we use a multi-step policy to decay the learning rate. We run a total of 290 epochs. We reduce the initial learning rate by 1/10 at the 145-th and 220-th epoch. For the Sync method, we are able to scale the batch size to 1K without losing accuracy (Table 3). We did not use data augmentation in this experiment. The accuracy of the original DenseNet implementation without data augmentation is 92.91%[4].

[4] github.com/liuzhuang13/DenseNetCaffe

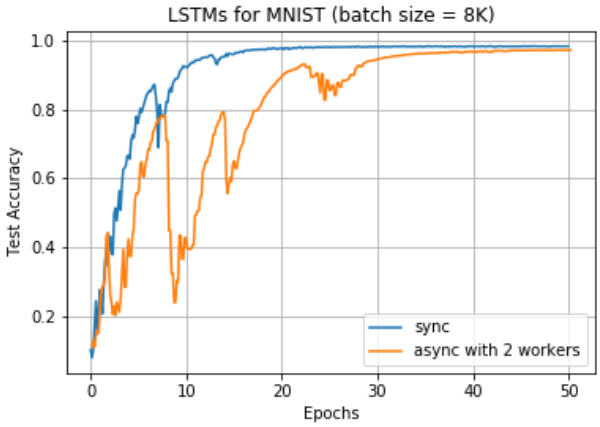

Figure 5: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size of 8192. The Sync method slightly beats the Async EA-wild method (two workers).

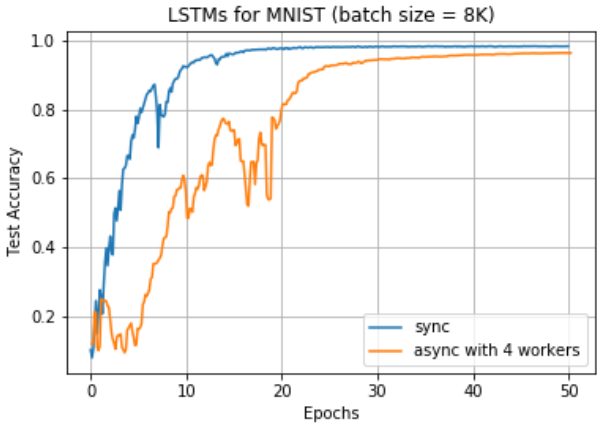

Figure 6: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size of 8192. The Sync method slightly beats the Async EA-wild method (four workers).

The Async EA-wild method saw about a 3% accuracy drop. We believe we searched the hyper-parameters comprehensively in the tuning space. The Sync method uses 16 machines, while the Async EA-wild method uses one server and 16 workers (17 machines). In this example, the Sync method beats the Async EA-wild method with regards to accuracy. However, with regards to system speed, the Async EA-wild method on 17 KNLs is 1.11× faster than the Sync method on 16 KNLs for running the same 290 epochs.

4.5.5 GoogleNet model for ImageNet dataset. We use GoogleNet-v2, which is an implementation of the GoogleNet model with batch normalization. In this experiment, the weight decay is 0.0002, the

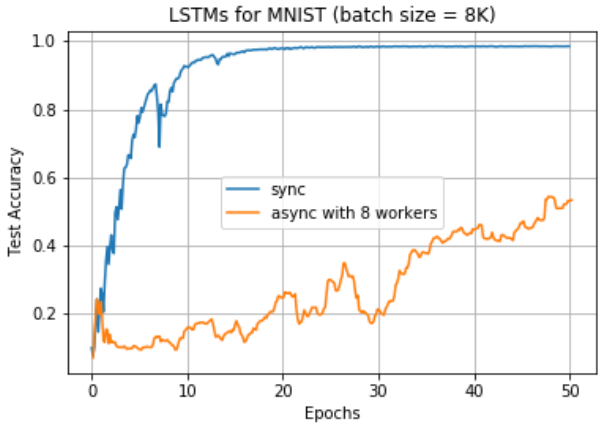

Figure 7: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size is 8192. The Sync method clearly outperforms the Async EA-wild method (eight workers).

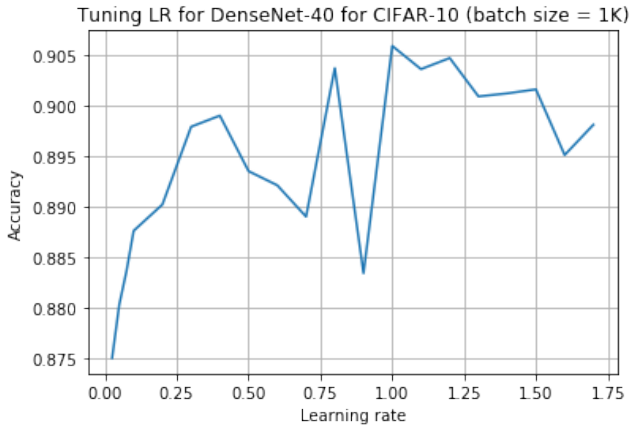

Figure 8: Tuning the learning rate for CIFAR-10 with the DenseNet-40 model, using the EA-wild asynchronous solver with a batch size of 1K. 290 epochs total, weight decay is 0.0001, and momentum is 0.9.

momentum is 0.9, and we use a polynomial policy to decay the learning rate (power = 0.5). Using the Sync method, we get 71.27% top-1 accuracy (without data augmentation) in 72 epochs by a batch size of 1024. We did not use data augmentation in this experiment. The accuracy of Caffe's implementation without data augmentation is 68.7%[5]. For the Async EA-wild method, the number of workers has an influence on the accuracy. In Table 4 we observe that using more workers for the same batch size will lead to accuracy decay. However, even when we only use 8 workers, the accuracy of the Async EA-wild method is still lower than that of the Sync method

[5] github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

Table 3: Accuracy for CIFAR-10 with DenseNet-40 model. 290 epochs total, weight decay is 0.0001, and momentum is 0.9. Here we compare the Sync method against the tuned asynchronous EA-wild method and observe that the Sync method outperforms it in terms of accuracy.

Batch Size	Method	LR	warmup	Accuracy
64	Sync	0.1	0 epochs	92.91%
1K	Sync	0.8	15 epochs	94.15%
1K	Async EA-wild	1.0	15 epochs	90.59%

Table 4: Accuracy for ImageNet with GoogleNet model. 72 epochs total, weight decay is 0.0002, and momentum is 0.9. S means server machine and W means worker machine. The Async method refers specifically to the EA-wild solver. The Sync method beats the Async EA-wild method.

Batch Size	Method	LR	machines	Accuracy
1K	Sync	0.12	64 Ws	71.26%
1K	Async	0.005	1 S + 16 Ws	58.68%
1K	Async	0.01	1 S + 16 Ws	61.87%
1K	Async	0.02	1 S + 16 Ws	62.49%
1K	Async	0.04	1 S + 16 Ws	61.40%
1K	Async	0.06	1 S + 16 Ws	59.26%
1K	Async	0.08	1 S + 16 Ws	57.41%
1K	Async	0.10	1 S + 16 Ws	55.58%
1K	Async	0.12	1 S + 16 Ws	52.34%
1K	Async	0.06	1 S + 8 Ws	65.68%
1K	Async	0.06	1 S + 16 Ws	59.26%
1K	Async	0.06	1 S + 32 Ws	50.03%
1K	Async	0.06	1 S + 64 Ws	35.69%

(65.68% vs 71.27%). If the Async EA-wild method uses 64 workers, the accuracy is only 35.69%. After comprehensive parameter tuning, we conclude that the Sync method beats the Async EA-wild method with regards to accuracy for ImageNet training with GoogleNet. With regards to system speed, the Sync method on 16 KNLs is 1.33× faster than the Async EA-wild method on 17 KNLs for running the same 72 epochs.

4.5.6 ResNet-50 model for ImageNet dataset. With weak data augmentation (1-crop, center 224x224 crop from resized image with shorter side=256), the accuracy of original ResNet-50 is 75.3%[6]. With stronger data augmentation, ResNet-50-v2 can achieve 76.2% accuracy. In this paper, we use the original version of ResNet-50. Using the Sync method, we get 75.3% accuracy after 90 epochs with a batch size of 8192. In this experiment, the weight decay is 0.0001, the momentum is 0.9, and we use a multi-step policy to decay the learning rate. We run 90 epochs total. We reduce the initial learning rate by 1/10 at the 30-th, 60-th and 80-th epoch. We use the 5-epoch warmup scheme for learning rate[6]. From Table 5, we observe the same pattern with the GoogleNet case. Although tuning the learning rate and reducing the number of workers can help to improve the accuracy, the accuracy of the Async EA-wild method is much

[6] github.com/KaimingHe/deep-residual-networks

Table 5: Accuracy for ImageNet with ResNet-50 model. 90 epochs total, weight decay is 0.0002, and momentum is 0.9. S means server machine and W means worker machine. The Async method refers specifically to the EA-wild solver. The Sync method beats the Async EA-wild method.

Batch Size	Method	LR	machines	Accuracy
256	Sync	0.1	16 Ws	75.30%
8192	Sync	3.2	512 Ws	75.30%
8192	Async	0.2	1 S + 64 Ws	26.87%
8192	Async	0.4	1 S + 64 Ws	28.69%
8192	Async	0.8	1 S + 64 Ws	26.70%
8192	Async	1.6	1 S + 64 Ws	14.36%
8192	Async	3.2	1 S + 64 Ws	3.35%
2048	Async	0.8	1 S + 64 Ws	3.39%
2048	Async	0.4	1 S + 64 Ws	19.98%
512	Async	0.2	1 S + 32 Ws	35.68%
512	Async	0.05	1 S + 32 Ws	43.80%
512	Async	0.01	1 S + 32 Ws	39.68%
256	Async	0.1	1 S + 16 Ws	47.51%

lower than that of the Sync method (28.69% vs 75.30%). Even if we use the baseline batch size (i.e. 256), the Sync method's accuracy is still much higher than that of the Async EA-wild method (47.51% vs 75.30%). In this case, the Sync method beats the Async EA-wild method. With regards to system speed, the Async EA-wild method on 65 KNLs is 1.41× faster than the Sync method on 64 KNLs when running the same 90 epochs.

4.6 Computation-Communication Ratio

To more easily analyze our results, we define a value which serves as a measure of a model's complexity compared with its size. This can be expressed as a computation-communication ratio (CC ratio). The measure of communication, or transferred data volume, at each iteration is proportional to the gradient size. In other words, it represents the model size or the number of parameters in the model. On the other hand, the measure of computation is proportional to the number of floating point operations conducted for processing one sample.

For example, the model size of ResNet-50 is around 100 MB, with 25 million parameters. Processing one ImageNet sample with ResNet-50 requires 7.7 billion operations. Thus, the CC ratio of ResNet-50 is 308. Similarly, the CC ratio of GoogleNet is 736 (with 9.7 billion operations and 13.5 million parameters). The DenseNet-40 model [8] has 9 million parameters and requires about 5 billion operations, for a CC ratio of 555. Since we do not have the exact number of operations required for the cifar10-full model, we can estimate the CC ratio of the cifar10-full model by looking at the widely used AlexNet model [9]. The AlexNet model, which is very similar to the cifar10-full model, has 62.25 million parameters and requires 6.8 billion operations. Its CC ratio is 109, lower than any of the other models we have tested.

By inspecting the CC ratio of each of our experiments above, we can conclude that the Sync solver is more suitable for models with a high CC ratio, or models which are more computationally intensive.

Table 6: Summary of Sync vs. Async EA-wild for different models. The accuracy of each solver uses the same batch size in order to keep the comparison fair. The best performance of each synchronous and asynchronous solver is selected.

Model	CC ratio	Baseline	Sync	Async EA-wild
cifar10-full	~ 109	82%	74.92%	75.51%
DenseNet-40	555	93%	94.15%	90.59%
ResNet-50	308	75.3%	75.30%	47.51%
GoogLeNet	736	68.7%	71.26%	65.68%

Table 7: Summary of Sync vs. Async EA-wild speed comparisons for different models. For the number of machines used for the Async EA-wild method, add one to the number of Sync KNLs denoted in the table.

Model	# Sync KNLs	Epochs	Speedup
cifar10-full	8	140	Sync is 1.44x faster
DenseNet-40	16	290	Async is 1.11x faster
ResNet-50	64	90	Async is 1.41x faster
GoogLeNet	16	72	Sync is 1.33x faster

5 SUMMARY AND CONCLUSION

We use four real-world applications to conduct a comparison between the Sync and Async EA-wild methods. It is worth noting that the epoch-accuracy relationship of our comparison is not dependent on hardware. The analysis is based on numerical results, which are only dependent on the algorithm, data and model; we expect to get similar results across multiple types of processors such as KNLs, TPUs or GPUs.

From the results above, we observe that the Async EA-wild method only slightly beats the Sync method for the cifar10-full model with regards to system speed. The cifar10-full model is a naive model while LSTM, DenseNet, GoogleNet, and ResNet-50 are more complex models. In the large batch situation, we observe that the Sync method tends to outperform the Async EA-wild method when dealing with more computationally intensive models with a high CC ratio. With regards to the system speed comparison, the Sync method is faster than the Async EA-wild method for LSTM, cifar10-full and GoogleNet models, while the Async EA-wild method is faster than the Sync method for DenseNet-40 and ResNet-50. Empirically, these results lead us to conclude that synchronous solvers may be more suited for machine learning applications that require more complex models. Asynchronous solvers are more unstable as the gradient updates are delayed, and thus may be more suited for models with a low CC ratio.

6 FUTURE WORK

Future areas of exploration include analyzing the trade-offs between asynchronous and synchronous solvers for reinforcement learning applications. There is an existing body of work exploring asynchronous methods for deep reinforcement learning [15] [16]

[3], and an area of interest is investigating how these methods might be able to scale up on supercomputing clusters. Such future investigations might yield insights into the trade-offs of asynchronous and synchronous solvers for the more general learning tasks that reinforcement learning addresses.

Another promising area of future research lies in exploring asynchronous approaches for federated learning. The issues of privacy and user data confidentiality have emerged as crucial topics for machine learning, as traditional machine learning systems require training data to be aggregated into a centralized data center. Such centralized systems can lead to privacy concerns. Federated learning takes a decentralized approach to machine learning and enables users to collaboratively learn a model while keeping sensitive data private. Previous work on Layer-wise Adaptive Rate Scaling (LARS) [24] is currently powering some of the federated learning projects at Google. An investigation into how to design algorithms that minimize the communication needed to train models with federated learning would enable its usage for more generalized applications.

7 ACKNOWLEDGEMENTS

We would like to thank Professor Cho-Jui Hsieh at UCLA for his helpful discussions with us regarding the theoretical analysis of our EA-wild algorithm. We would also like to thank CSCS, the Swiss National Supercomputing Centre, for allowing us access to the Piz Daint supercomputer, on which we ran our experiments comparing synchronous and asynchronous solvers.

REFERENCES

[1] T. Akiba, S. Suzuki, and K. Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.
[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.
[3] W. M. Czarnecki, R. Pascanu, S. Osindero, S. M. Jayakumar, G. Swirszcz, and M. Jaderberg. Distilling Policy Distillation. *arXiv e-prints*, page arXiv:1902.02186, Feb 2019.
[4] 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*, pages 1223–1231, 2012.
[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.
[6] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
[8] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.
[9] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.
[10] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.
[11] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.
[12] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, pages 2331–2339. Curran Associates Inc., 2009.
[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
[14] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama, et al. Imagenet/resnet-50 training in 224 seconds. *arXiv preprint arXiv:1811.05233*, 2018.
[15] V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1602.01783, Feb 2016.
[16] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1507.04296, Jul 2015.
[17] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. 2011.
[18] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.
[19] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.
[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
[22] M. Yamazaki, A. Kasagi, A. Tabuchi, T. Honda, M. Miwa, N. Fukumoto, T. Tabaru, A. Ike, and K. Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.
[23] Y. You, A. Buluç, and J. Demmel. Scaling deep learning on gpu and knights landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2017.
[24] Y. You, I. Gitman, and B. Ginsburg. Large Batch Training of Convolutional Networks. *arXiv e-prints*, page arXiv:1708.03888, Aug 2017.
[25] Y. You, I. Gitman, and B. Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 2017.
[26] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes. *CoRR, abs/1709.05011*, 2017.
[27] S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.