Efficient Parallel Computing for Machine Learning at Scale

Arissa Wongpanich

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-225
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-225.html

December 18, 2020

Efficient Parallel Computing for Machine Learning at Scale

by

Arissa Wongpanich

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Chair
Professor Joseph Gonzalez

Fall 2020

Efficient Parallel Computing for Machine Learning at Scale

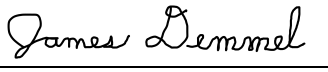
by Arissa Wongpanich

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

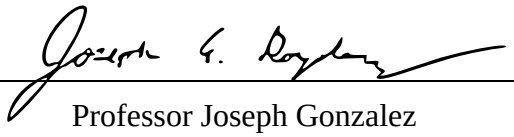


Professor James Demmel
Research Advisor

12/17/2020

(Date)

* * * * * * *



Professor Joseph Gonzalez
Second Reader

12/17/2020

(Date)

Abstract

Efficient Parallel Computing for Machine Learning at Scale

by

Arissa Wongpanich

Masters of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor James Demmel, Chair

Recent years have seen countless advances in the fields of both machine learning and high performance computing. Although computing power has steadily increased and become more available, many widely-used machine learning techniques fail to take full advantage of the parallelism available from large-scale computing clusters. Exploring techniques to scale machine learning algorithms on distributed and high performance systems can potentially help us reduce training time and increase the accessibility of machine learning research. To this end, this thesis investigates methods for scaling up deep learning on distributed systems using a variety of optimization techniques, ranging from clusters of Intel Xeon Phi processors to Tensor Processing Unit (TPU) pods. Training machine learning models and fully optimizing compute on such distributed systems requires us to overcome multiple challenges at both the algorithmic and the systems level. This thesis evaluates and presents scaling methods for distributed systems which can be used to address such challenges, and more broadly, to bridge the gap between high performance computing and machine learning.

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful for all of the help and support I have received from my peers, friends, family, and mentors during my academic journey at Berkeley. I would first like to thank my advisor, Professor Jim Demmel, for his guidance, mentorship, and kindness over the past few years, without which this thesis would not have been possible. I am inspired by his dedication to his research, his commitment to mentorship, and his passion for the field, and I have learned so much from him about the worlds of high performance computing, communication-avoiding algorithms, and beyond. I am honored to have had the opportunity to work with him and all of the amazing people at Berkeley in the BeBOP group and ADEPT lab.

I would also like to thank Professor Yang You, currently at the National University of Singapore, for his mentorship and advice. I feel very lucky to have had the opportunity to collaborate with him on many projects during my undergraduate and graduate studies at Berkeley, and have learned an immense amount from him.

I want to extend my deepest gratitude to Professor Joey Gonzalez for his research advice and for sharing with me his enthusiasm for systems and machine learning. I also thank him for his feedback as my second thesis reader.

Some of the research included in this thesis was conducted during my internship at Google Brain. I give special thanks to the TPU Performance team at Google, in particular to Sameer Kumar, who was my mentor and who provided invaluable guidance and assistance with my work. I would also like to thank my other co-authors at Google: Hieu Pham, Mingxing Tan, and Quoc Le.

Finally, I would like to thank my friends and family for their support.

Chapter 1

Introduction

1.1 Motivation

Machine learning has become ubiquitous in recent years, with applications spanning the domains of object detection [12], language modeling [38], speech recognition [14], recommendation systems [31], and more. These advances have largely been powered by larger datasets, more available computing power, and innovations in both software and hardware. Despite these gains, many current state-of-the-art models still take on the order of days or months to train and must utilize enormous amounts of compute. For example, OpenAI's Generative Pre-trained Transformer 3 (GPT-3) model, with 175 billion parameters, requires compute on the order of several thousand petaflop/s-days (pfs-day), a unit consisting of 10^{15} neural net operations per second per day [5]. Such compute-intensive models have become a considerable bottleneck among researchers and machine learning practitioners. In addition, although computing power has steadily increased and become more available, many widely-used machine learning techniques fail to take full advantage of the parallelism available from these computing clusters. Exploring techniques to scale machine learning algorithms on distributed and high performance systems can potentially help us tackle this problem and increase the pace of development as well as the accessibility of machine learning research.

To this end, this thesis investigates methods for scaling up deep learning on distributed systems using a variety of optimization techniques, ranging from clusters of Intel Xeon Phi processors to Tensor Processing Unit (TPU) pods. Training machine learning models and fully optimizing compute on such distributed systems requires us to overcome multiple challenges at both the algorithmic and the systems level. Some of these challenges include maintaining accuracy when training with extremely large batch sizes, optimizing operations to run efficiently on distributed processors, and selecting the most optimal machine learning algorithms to train a given model. This thesis evaluates and presents scaling methods for distributed systems which can be used to address such challenges, and more broadly, to bridge the gap between high performance computing and deep learning.

1.2 Contributions

Exploring Asynchronous and Synchronous Machine Learning Algorithms at Scale

In Chapter 2, 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! [30] with EASGD. Our implementation is able to scale to 217,600 cores and finish 90 epochs of ResNet-50 training on ImageNet in 15 minutes. For comparison, 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.

Methods for Petascale Image Classification

In Chapter 3, we look at scaling up training on TPU Pods, specifically focusing on EfficientNet, a state-of-the-art image classification model based on efficiently scaled convolutional neural networks. Currently, EfficientNets can take on the order of days to train; for example, training an EfficientNet-B0 model takes 23 hours on a Cloud TPU v2-8 node [36]. Motivated by speedups that can be achieved when training at such scales, we explore techniques to scale up the training of EfficientNets on TPU-v3 Pods with 2048 cores. We discuss optimizations required to scale training to a batch size of 65536 on 1024 TPU-v3 cores, such as selecting large batch optimizers and learning rate schedules, as well as utilizing distributed evaluation and batch normalization techniques. Additionally, we present timing and performance benchmarks for EfficientNet models trained on the ImageNet dataset in order to analyze the behavior of EfficientNets at scale. With our optimizations, we are able to train EfficientNet on ImageNet to an accuracy of 83% in 1 hour and 4 minutes, demonstrating that our techniques are effective for scaling up training on peta-scale computing systems.

Chapter 2

Exploring Asynchronous and Synchronous Machine Learning Algorithms at Scale

2.1 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 [3] have come at the cost of longer training time. As a result, state-of-the-art models like OpenAI's GPT-3 [5] or AlphaZero [33] 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 chapter, 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 chapter. 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 [9]. 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 [20].

For the asynchronous algorithms, we first study Elastic Averaging Stochastic Gradient Descent (EASGD) [49] since its scalability can potentially be improved. In the original EASGD algorithm, 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 [30] 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.

Notation. Throughout this chapter, 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.

2.2 Background and Related Work

2.2.1 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 chapter we focus on data parallelism.

2.2.2 Recent Hardware for Distributed Learning

Researchers commonly train their models on servers with multiple GPUs, such as NVIDIA's P100 and V100 GPUs; on the other hand, recently developed many-core chips such as Google's Tensor Processing Units (TPUs) and Intel Knights Landing (KNL) chips also offer good performance. Researchers can use clusters 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 state-of-the-art models since 2018 (e.g. Google BERT, OpenAI GPT-3, AlphaZero) have been trained on such large-scale clusters. To make full use of these massively parallel hardware resources, efficient parallel and distributed solvers are necessary.

2.2.3 State-of-the-Art Asynchronous Solvers

Since the first large-scale asynchronous solver was implemented by Google Brain [9], several different asynchronous solvers have been used in industry. The most widely-used methods are Parameter-Servers, Hogwild! SGD and EASGD.

2.2.3.1 Hogwild! SGD

Classical Stochastic Gradient Descent (SGD) uses a lock between different weight updates to avoid thread conflicts [23]. 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 [30], 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 proven [30] that the algorithm still converges to the optimum under certain assumptions.

2.2.3.2 Elastic Averaging SGD (EASGD)

The EASGD method [49] has been proposed as a variant of SGD for distributed systems. EASGD formulates the problem as

$$\min_{\boldsymbol{w}^1,\ldots,\boldsymbol{w}^P,\tilde{\boldsymbol{w}}} \sum_{j=1}^{P} (f_j(\boldsymbol{w}^j) + \frac{\rho}{2}\|\boldsymbol{w}^j - \tilde{\boldsymbol{w}}\|_2^2), \tag{2.1}$$

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 (2.1), Zhang et al. [49] has proposed the following scheme. Local workers conduct the local SGD updates with respect to $\boldsymbol{w}^j$:

$$\boldsymbol{w}_{t+1}^j = \boldsymbol{w}_t^j - \eta(\nabla f_j(\boldsymbol{w}_t^j) + \rho(\boldsymbol{w}_t^j - \widetilde{\boldsymbol{w}}_t)), \tag{2.2}$$

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 [49].

Each update to the global parameter can be written as

$$\tilde{\boldsymbol{w}}_{t+1} = \tilde{\boldsymbol{w}}_t + \eta\rho(\boldsymbol{w}_t^j - \tilde{\boldsymbol{w}}_t). \tag{2.3}$$

The ρ in Equation (2.2) and Equation (2.3) 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.

2.3 Implementing Asynchronous and Synchronous solvers

2.3.1 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 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 [44]. We additionally provide a strong theoretical guarantee to EA-wild (Section 2.3.3).

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

2.3.2 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 use 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.

2.3.3 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 [30], we refer the reader to the proof in the original chapter [29]. 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 chapter [49].

We restate the convergence bounds derived in the Hogwild! chapter [29], based on the following finite sum problem:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \quad f(\boldsymbol{w}) := \sum_{e\in E} f_e(\boldsymbol{w}_e), \tag{2.4}$$

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 (2.4) 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:

$$\Omega := \max_{e\in E}|e|, \tag{2.5}$$

$$\Delta := \frac{\max_{1\le v\le n}|\{e \in E : v \in e\}|}{|E|}, \tag{2.6}$$

$$\rho := \frac{\max_{e\in E}|\{\hat{e} \in E : \hat{e} \cap e \neq \phi\}|}{|E|}. \tag{2.7}$$

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! [29].

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*

$$\gamma = \frac{\theta\epsilon c}{2LM^2(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{\frac{1}{2}})} \tag{2.8}$$

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

$$k \ge 2LM^2(1 + 6\rho\tau + 6\tau^2\Omega\Delta^{\frac{1}{2}})\frac{\log(LD_0/\epsilon)}{c^2\theta\epsilon}. \tag{2.9}$$

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 [29], 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.

2.4 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 2.4.5 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.

2.4.1 Experimental Settings

Accuracy in this chapter 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 chapter include the MNIST [24] dataset, CIFAR-10 [21], and ImageNet [10]. The MNIST dataset is processed by the LeNet model [24] and a pure-LSTM model. The CIFAR-10 dataset is processed by the DenseNet model [15]. The ImageNet dataset is processed by the GoogLeNet model [34] and the ResNet-50 model [13].

2.4.2 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 [30].
- 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.

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.

2.4.3 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 2.1. 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.

2.4.4 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 2.2. 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.

2.4.5 Async vs Sync SGD for large-batch training

For state-of-the-art deep neural network training, some researchers use asynchronous methods [9] while other researchers prefer synchronous methods [11]. Since large-batch methods can improve performance, they have recently been actively studied [2], [11], [25], [42], [48]. 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

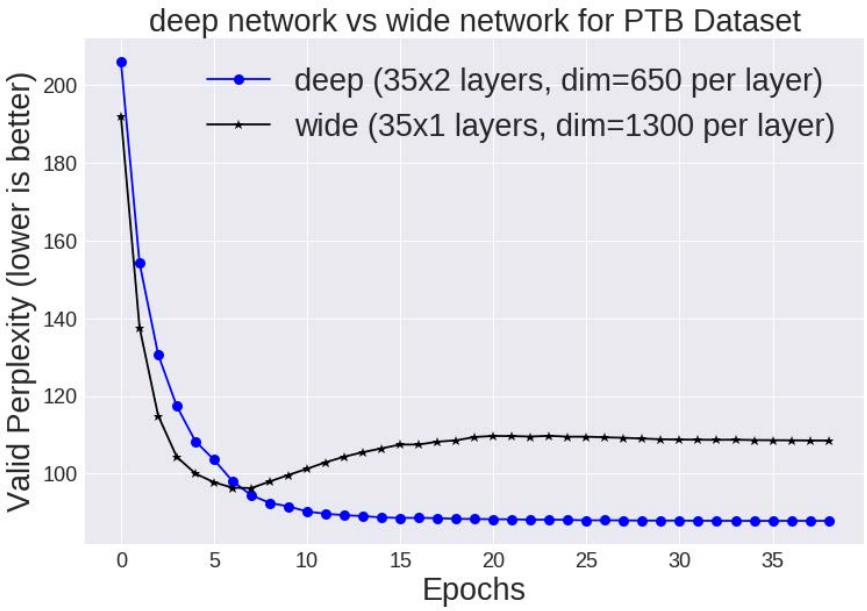


Figure 2.1: Given the same number of parameters, we observe that deep neural networks constantly beat wide neural networks. We use this information to maximize the parallelism in distributed training.

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 2.1.

2.4.5.1 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 [20]. To minimize the accuracy loss of async solvers, we design a two-stage learning rate scheduling (Figure 2.3).

2.4.5.2 CIFAR10-full model for CIFAR-10 Dataset.

CIFAR10-full is implemented by the Caffe team for fast CIFAR-10 training. In this experiment, we set weight decay to 0.004 and momentum to 0.9, and we use a polynomial policy to decay the learning rate (power = 1.0). As suggested by earlier work [11], 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[1]. By utilizing a warmup scheme and the

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

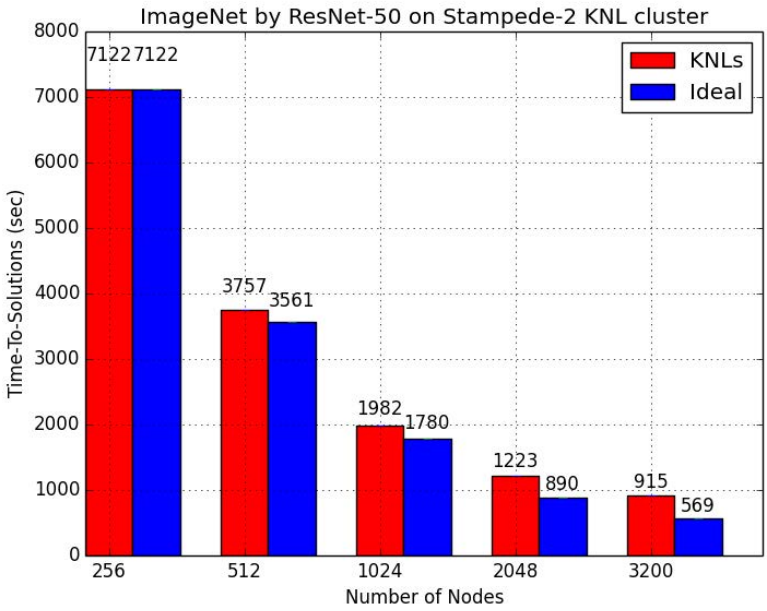


Figure 2.2: We scaled the DNN training to 217,600 cores (each KNL has 72 CPU cores) and finished the 90-epoch ImageNet training with ResNet-50 model in 15 minutes using the EA-wild asynchronous algorithm. Ideal training time is computed by simply dividing the total training time by the number of workers.

LARS [46] optimizer, we are able to scale the batch size to 5K with a reasonable accuracy (Table 2.2). However, 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 2.4). 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.

2.4.5.3 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

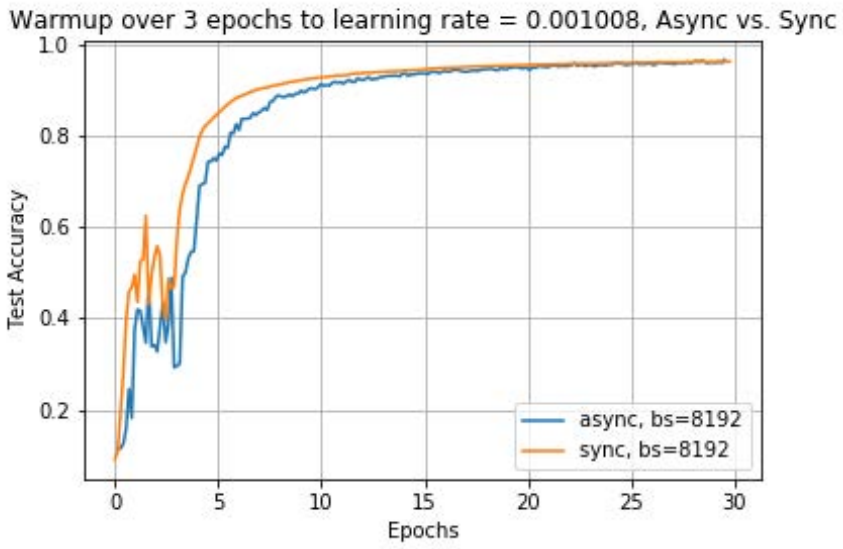


Figure 2.3: We use Gradient Descent Optimizer during the warmup and then Adam Optimizer once warmup is complete. We chose to use Gradient Descent Optimizer during the warmup because we can easily set the learning rate, whereas Adam Optimizer can dynamically change the learning rate based on the online 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 2.1: 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%

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, 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 2.5-2.7). As we increase the number of workers, the performance of async solvers degrades.

Table 2.2: 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%

Table 2.3: 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%

2.4.5.4 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 2.3). We did not use data augmentation in this experiment. The accuracy of the original DenseNet implementation without data augmentation is 92.91%[2]. 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.

[2]github.com/liuzhuang13/DenseNetCaffe

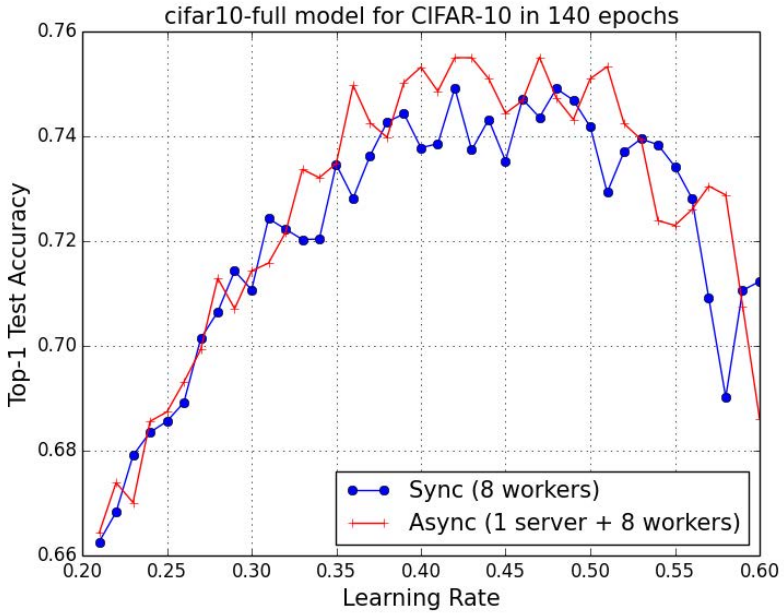


Figure 2.4: Accuracy for cifar10-full model. 140 epochs total, weight decay is 0.004, momentum is 0.9, and batch size is 8192. The Async EA-wild method beats the Sync method with respect to accuracy.

2.4.5.5 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 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%[3]. For the Async EA-wild method, the number of workers has an influence on the accuracy. In Table 2.4 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 (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

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

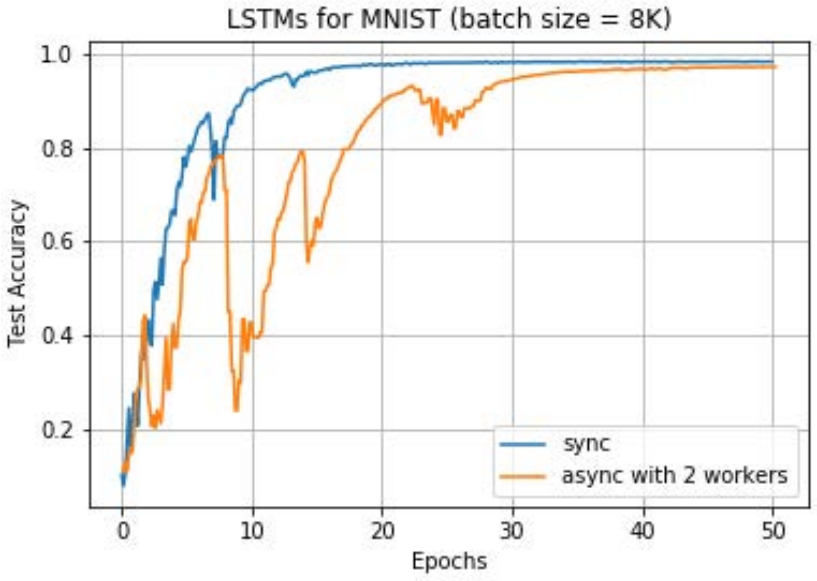


Figure 2.5: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size of 8192. The Sync method slightly beats the Async EA-wild method (two workers).

than the Async EA-wild method on 17 KNLs for running the same 72 epochs.

2.4.5.6 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%[4]. With stronger data augmentation, ResNet-50-v2 can achieve 76.2% accuracy. In this chapter, 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[11]. From Table 2.5, 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 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

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

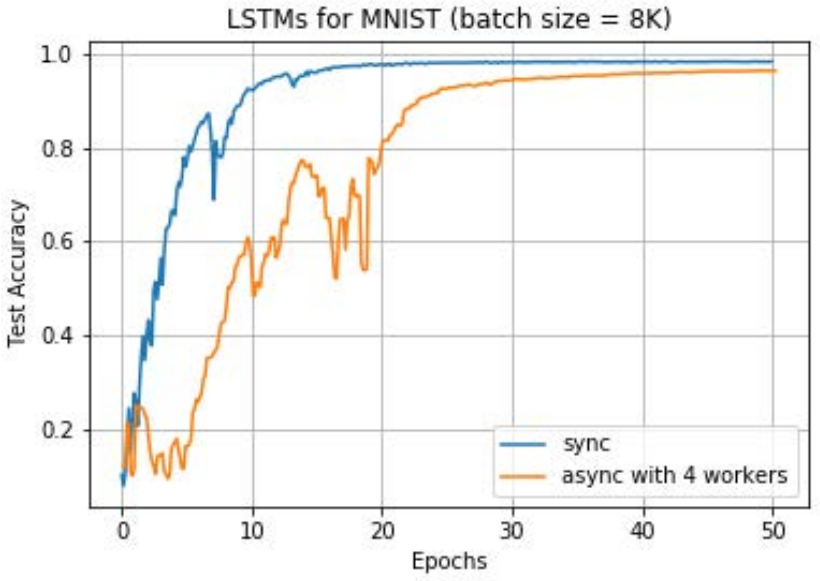


Figure 2.6: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size of 8192. The Sync method slightly beats the Async EA-wild method (four workers).

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.

2.4.6 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 [15] has 9 million parameters and requires about 5 billion operations, for a CC ratio of 555. Since we do not

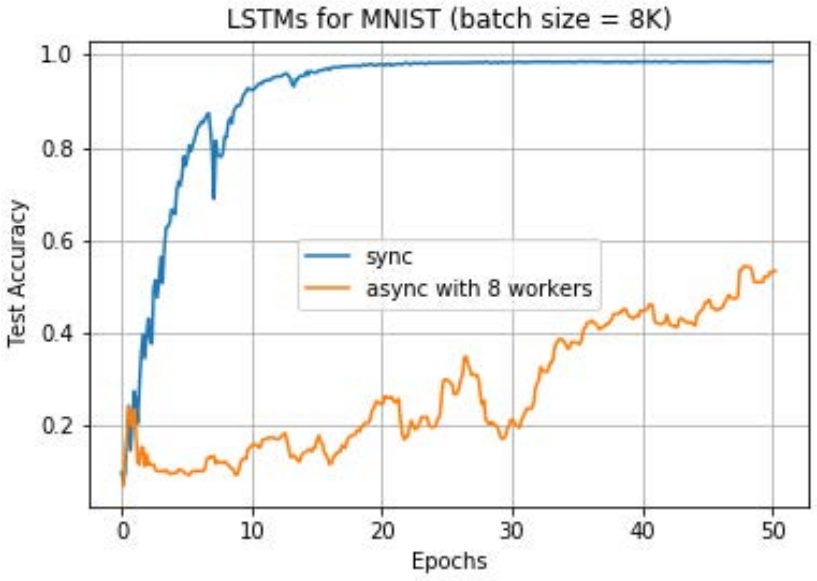


Figure 2.7: Accuracy for MNIST with LSTMs. 50 epochs total, Adam optimizer for learning rate tuning, and batch size is 8192. The Sync method clearly outperforms the Async EA-wild method (eight workers).

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 [16]. 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.

2.5 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.

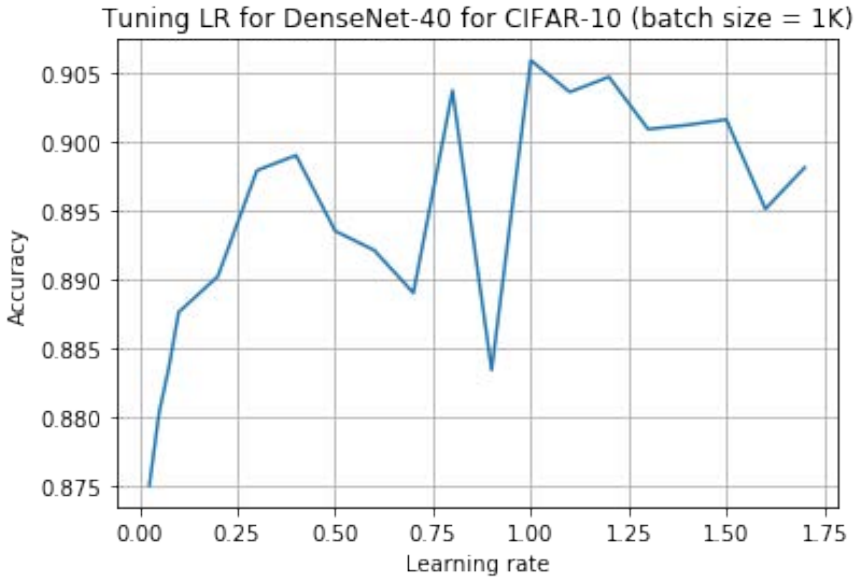


Figure 2.8: Tuning the learning rate for CIFAR-10 with the DenseNet-40 model, using the EA-wild asynchronous solver with a batch size of 1K. 290 epochs total, weight decay is 0.0001, and momentum is 0.9.

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.

2.6 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

Table 2.4: 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%

work exploring asynchronous methods for deep reinforcement learning [27, 28, 8], 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) [45] 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.

Table 2.5: 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%

Table 2.6: 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 2.7: 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

Chapter 3

Methods for Petascale Image Classification

3.1 Introduction

As machine learning models have gotten larger [5], so has the need for increased computational power. Large clusters of specialized hardware accelerators such as GPUs and TPUs can currently provide computations on the order of petaFLOPS, and have allowed researchers to dramatically accelerate training time. For example, the commonly used ResNet-50 image classification model can be trained on ImageNet [32] in 67 seconds on 2048 TPU cores [43], a substantial improvement from a typical training time taking on the order of hours. In order to accelerate training of machine learning models with petascale compute, large-scale learning techniques as well as specialized systems optimizations are necessary.

EfficientNets [35], a family of efficiently scaled convolutional neural nets, have recently emerged as state-of-the-art models for image classification tasks. EfficientNets optimize for accuracy as well as efficiency by reducing model size and floating point operations executed while still maintaining model quality. Training an EfficientNet-B0 model on Cloud TPU v2-8, which provides 8 TPU-v2 cores, currently takes 23 hours [36]. By scaling up EfficientNet training to a full TPU-v3 pod, we can significantly reduce this training time.

Training at such scales requires overcoming both algorithmic and systems-related challenges. One of the main challenges we face when training at scale on TPU-v3 Pods is the degradation of model accuracy with large global batch sizes of 16384 or greater. Additionally, the default TensorFlow APIs for TPU, TPUEstimator [37], constrains evaluation to be performed on a separate TPU chip, thereby creating a new compute bottleneck from the evaluation loop [1, 6]. To address these challenges, we draw from various large-scale learning techniques, including using an optimizer designed for training with large batch sizes, tuning learning rate schedules, distributed evaluation, and distributed batch normalization. With our optimizations, we are able to scale to 1024 TPU-v3 cores and a batch size of 65536 to reduce EfficientNet training time to one hour while still achieving 83% accuracy. We discuss

our optimizations in Section 3.3 and provide analysis and benchmarks of our results in Section 3.4.

3.2 Related Work

Training machine learning models with more TPU cores requires increasing the global batch size to avoid under-utilizing the cores. This is because the TPU cores operate over a memory layout of XLA [41], which pads each tensor's batch dimension to a multiple of eight [18]. When the number of TPU cores increases to the point that each core processes fewer than 8 examples, the cores will have to process the padded examples, thus wasting resources. Therefore, training on an entire TPU-v3 pod which has 2048 TPU cores requires at least a global batch size of 16384.

It has been observed that when training with such large batch sizes there is degradation in model quality compared to models trained with smaller batch sizes due to a "generalization gap" [20]. Previous works on large-batch training have addressed this issue using an amalgamation of techniques, such as:

- Adjusting the learning rate scaling and warm-up schedules [11]
- Adjusting the computation of batch-normalization statistics [2, 11]
- Using optimizers that are designed for large batch sizes, such as LARS [45] or SM3 [4]

Together, these techniques have allowed training ResNet-50 on ImageNet in 2.2 minutes [43], BERT in 76 minutes [47], and more recently ResNet-50 on ImageNet in under 30 seconds [26], all without any degradation in model quality.

Despite all the impressive training time measures, we observe that in the image domain, these scaling techniques have merely been applied to ResNets. Meanwhile, these techniques have not been applied to EfficientNet despite their state-of-the-art accuracy and efficiency.

3.3 Methods

Scaling EfficientNet training to 1024 TPU-v3 cores introduces many challenges which must be addressed with algorithmic or systemic optimizations. The first challenge we face is maintaining model quality as the global batch size increases. Since the global batch size scales with the number of cores used for training, we must utilize large batch training techniques to maintain accuracy. We also face compute bottlenecks when training across large numbers of TPU chips, which we address using the distributed evaluation and batch normalization techniques presented in Kumar et al. [22]. The optimization techniques we explore to scale EfficientNet training on TPU-v3 Pods are described below:

3.3.1 Large batch optimizers

While the original EfficientNet paper used the RMSProp optimizer, it is known that with larger batch sizes RMSProp causes model quality degradation. We scale training to 1024 TPU-v3 cores via data parallelism, which means that the global batch size must scale up with the number of workers if we keep the per-core batch size fixed. For example, if we fix the per-core batch size at 32, the resulting global batch size on 1024 cores would be 32768. On the other hand, if the global batch size is fixed when scaling up to many cores, the resulting lower per-core batch size leads to inefficiencies and lower throughput. Thus, large global batch sizes are necessary for us to more optimally utilize the memory of each TPU core and increase throughput. Using the Layer-wise Adaptive Rate Scaling (LARS) optimizer proposed in You, Gitman, and Ginsburg [45], we are able to scale up to a batch size of 65536 while attaining similar accuracies as the EfficientNet baseline accuracies reported in Tan and Le [35].

3.3.2 Learning rate schedules

In order to maintain model quality at large batch sizes, we also adopt the learning rate warmup and linear scaling techniques described in [11]. Increasing the global batch size while keeping the number of epochs fixed results in fewer iterations to update weights. In order to address this, we apply a linear scaling rule to the learning rate for every 256 samples in the batch. However, larger learning rates can lead to divergence; thus, we also apply a learning rate warmup where training starts with a smaller initial learning rate and gradually increases the learning rate over a tunable number of epochs. In addition, we compared various learning rate schedules such as exponential decay and polynomial decay and found that for the LARS optimizer, a polynomial decay schedule achieves the highest accuracy.

3.3.3 Distributed evaluation

The execution of the evaluation loop is another compute bottleneck on the standard cloud TPU implementation of EfficientNet, since evaluation and training loops are executed on separate TPUs. With traditional TPUEstimator [37], where evaluation is carried out on a separate TPU, training executes faster than evaluation, causing the end-to-end time to depend heavily on evaluation time. To overcome this, we utilize the distributed training and evaluation loop described in Kumar et al. [22], which distributes training and evaluation steps across all TPUs and allows for scaling to larger numbers of replicas.

3.3.4 Distributed batch normalization

Additionally, we distribute the batch normalization across replicas by grouping subsets of replicas together, using the scheme presented in Ying et al. [43]. This optimization improves the final accuracy achieved with trade-offs on the communication costs between TPUs. The

number of replicas that are grouped together is a tunable hyperparameter. The resulting batch normalization batch size, which is the total number of samples in each replica subset, also affects model quality as well as convergence speed. For subsets of replicas larger than 16, we also explore a two-dimensional tiling method of grouping replicas together.

3.3.5 Mixed Precision

It has been observed that using the bfloat16 floating point format for training convolutional neural networks can match or even exceed performance of networks trained using traditional single precision formats such as fp32 [7, 17, 19], possibly due to a regularizing effect from the lower precision. We implement mixed-precision training to take advantage of the performance benefits of bfloat16 while still maintaining model quality. In our experiments, bfloat16 is used for convolutional operations, while all other operations utilize fp32. Using the bfloat16 format for convolutions improves hardware efficiency without degradation of model quality.

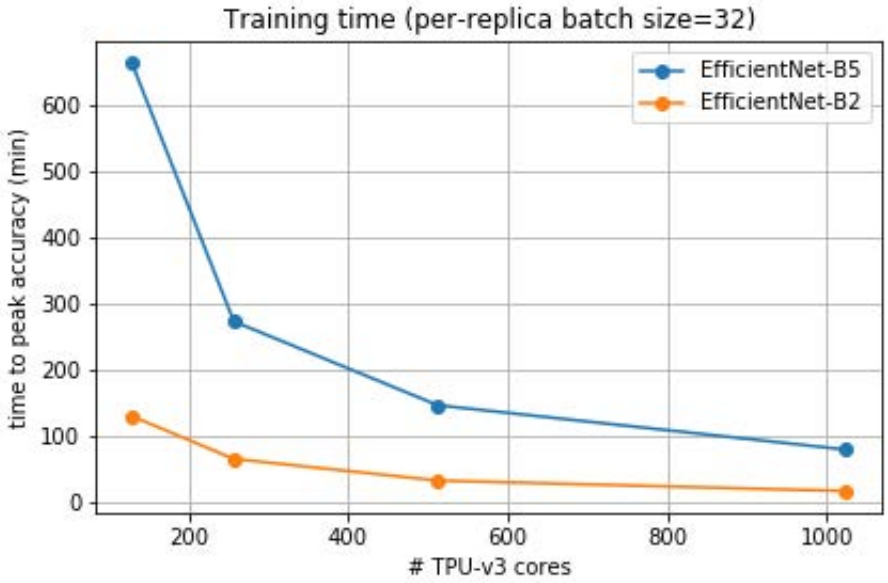


Figure 3.1: EfficientNet-B2 and B5 training time to peak accuracy for various TPU slice sizes. Training time starts immediately after initialization of the distributed training and evaluation loop and ends when the model reaches peak accuracy.

3.4 Results

In this section, we provide results from combining the techniques described above to train a variety of EfficientNet models on the ImageNet dataset at different TPU Pod slice sizes. We

train for 350 epochs to provide a fair comparison between the original EfficientNet baseline and our methods. We benchmark training times and accuracy by taking an average of three runs for each set of hyperparameters and configurations. The training time is measured by computing the time immediately after initialization of the distributed training and evaluation loop to the time when peak top-1 evaluation accuracy is achieved. As shown in Figure 3.1, we are able to observe a training time of 18 minutes to 79.7% accuracy for EfficientNet-B2 on 1024 TPU-v3 cores with a global batch size of 32768, representing a significant speedup. By scaling up the global batch size to 65536 on 1024 TPU-v3 cores, we are able to reach an accuracy of 83.0% in 1 hour and 4 minutes on EfficientNet-B5. The full benchmark of accuracies and respective batch sizes can be found in Table 3.2, demonstrating that with our methods we are able to maintain an accuracy of 83% on EfficientNet-B5 even as the global batch size increases.

Table 3.1: Comparison of communication costs and throughput on EfficientNet-B2 and B5 as the global batch size scales up.

Model	#TPU-v3 cores	Global batch size	Throughput (images/ms)	Percent of time spent on All-Reduce
EfficientNet-B2	128	4096	57.57	2.1
	256	8192	113.73	2.6
	512	16384	227.13	2.5
	1024	32768	451.35	2.81
EfficientNet-B5	128	4096	9.76	0.89
	256	8192	19.48	1.24
	512	16384	38.55	1.24
	1024	32768	77.44	1.03

Table 3.2: Benchmark of EfficientNet-B2 and B5 peak accuracies

Model	#TPU-v3 cores	Global batch size	Optimizer	Base LR	LR decay	LR warmup	Peak top-1 acc.
EfficientNet-B2	128	4096	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.801
	256	8192	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.800
	512	16384	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.799
	512	16384	LARS	15.102	Polynomial	50 epochs	0.795
	1024	32768	LARS	15.102	Polynomial	50 epochs	0.797
EfficientNet-B5	128	4096	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.835
	256	8192	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.834
	512	16384	RMSProp	0.016	Exponential over 2.4 epochs	5 epochs	0.834
	512	16384	LARS	0.236	Polynomial	50 epochs	0.833
	1024	32768	LARS	0.118	Polynomial	50 epochs	0.832
	1024	65536	LARS	0.081	Polynomial	43 epochs	0.830

Additionally, we provide a comparison of communication costs and throughput as the global batch size and number of TPU cores scales up in Table 3.1. We can see that as we increase the number of cores and thus the global batch size, the throughput scales up linearly. In other words, step time remains approximately the same at scale, which may be promising if we wish to scale up even further.

3.5 Future Work

We hope to conduct a deeper study on other large batch optimizers for EfficientNet, such as the SM3 optimizer [4], in an effort to further improve accuracy at large batch sizes. It has also been observed that batch normalization size can have an effect on accuracy [43], and there are various all-reduce schemes for distributed batch normalization that we can further investigate. In addition, model parallelism is a future area of exploration which would supplement the current data parallelism to allow training on large numbers of chips without standard global batch sizes.

Chapter 4

Conclusion and Technical Acknowledgements

This thesis studies a variety of methods for scaling up machine learning algorithms on distributed systems with a variety of accelerators ranging from CPUs (Intel Knights Landing) to TPUs, motivated by speedups that can be achieved when training at such scales. We present and evaluate both algorithmic and systems-level optimizations, such as using large batch training techniques to prevent accuracy degradation or modifying operations to run more efficiently in a distributed setting.

We also explore the importance of selecting the optimal machine learning algorithm for training certain models, and how the performance of such algorithms may be affected by model complexity. In particular, we observe that asynchronous algorithms such as EA-wild outperform synchronous methods when working with simpler models, but become highly unstable when dealing with more computationally complex models such as recurrent neural networks (RNNs). These results lead us to conclude that synchronous solvers may be more suited for machine learning applications that require more complex models.

Through our investigation of machine learning scaling techniques based on a variety of real-world workloads spanning vision and language domains, we aim to provide more insights into which methods work best and under what conditions. We hope our work can be a helpful reference in scaling up machine learning to distribued systems with peta-scale compute and beyond.

4.1 Technical Acknowledgements

This thesis is based on the following publications:

- Chapter 2 is based on a joint work with Yang You and James Demmel. It was published as a conference paper entitled *Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning* [40] in the International Conference on High Performance Computing in Asia-Pacific Region. (Wongpanich, You, Demmel, 2020).

- Chapter 3 is based on a joint work with Hieu Pham, James Demmel, Mingxing Tan, Quoc Le, Yang You, and Sameer Kumar, entitled *Training EfficientNets at Supercomputer Scale: 83% ImageNet Top-1 Accuracy in One Hour* [39]. (Wongpanich, Pham, Demmel, Tan, Le, You, Kumar, 2020).

Training EfficientNets at Supercomputer Scale: 83% ImageNet Top-1 Accuracy in One Hour is based on work done during an internship at Google Brain. My TPU experiments were carried out on Google's publicly available cloud TPU system and do not present any details related to Google's internal TPU system. I would like to thank my collaborators at both UC Berkeley and Google for their technical insights and advice.

For *Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning*, I would like to additionally thank Professor Cho-Jui Hsieh at UCLA for his helpful discussions with us regarding the theoretical analysis of our EA-wild algorithm. I 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.

Bibliography

[1] 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 Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv e-prints*, arXiv:1603.04467 (Mar. 2016), arXiv:1603.04467. arXiv: 1603.04467 [cs.DC].

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes". In: *arXiv preprint arXiv:1711.04325* (2017).

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. "Deep Speech 2: End-to-end Speech Recognition in English and Mandarin". In: *arXiv preprint arXiv:1512.02595* (2015).

[4] Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. "Memory-Efficient Adaptive Optimization". In: *arXiv e-prints*, arXiv:1901.11150 (Jan. 2019), arXiv:1901.11150. arXiv: 1901.11150 [cs.LG].

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[6] Heng-Tze Cheng, Zakaria Haque, Lichan Hong, Mustafa Ispir, Clemens Mewald, Illia Polosukhin, Georgios Roumpos, D Sculley, Jamie Smith, David Soergel, Yuan Tang, Philipp Tucker, Martin Wicke, Cassandra Xia, and Jianwei Xie. "TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks". In:

arXiv e-prints, arXiv:1708.02637 (Aug. 2017), arXiv:1708.02637. arXiv: 1708.02637 [cs.DC].

[7] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. "Learning Sparse Low-Precision Neural Networks With Learnable Regularization". In: *arXiv e-prints*, arXiv:1809.00095 (Aug. 2018), arXiv:1809.00095. arXiv: 1809.00095 [cs.CV].

[8] Wojciech Marian Czarnecki, Razvan Pascanu, Simon Osindero, Siddhant M. Jayakumar, Grzegorz Swirszcz, and Max Jaderberg. "Distilling Policy Distillation". In: *arXiv e-prints*, arXiv:1902.02186 (2019), arXiv:1902.02186. arXiv: 1902.02186 [cs.LG].

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. "Large Scale Distributed Deep Networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, pp. 248–255.

[11] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. *Mask R-CNN*. 2018. arXiv: 1703.06870 [cs.CV].

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[14] Yanzhang He, Tara N. Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, Qiao Liang, Deepti Bhatia, Yuan Shangguan, Bo Li, Golan Pundak, Khe Chai Sim, Tom Bagby, Shuo yiin Chang, Kanishka Rao, and Alexander Gruenstein. *Streaming End-to-end Speech Recognition For Mobile Devices*. 2018. arXiv: 1811.06621 [cs.CL].

[15] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. "Densely Connected Convolutional Networks". In: *arXiv preprint arXiv:1608.06993* (2016).

[16] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt 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*. 2016, pp. 2592–2600.

[17] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes". In: *arXiv e-prints*, arXiv:1807.11205 (July 2018), arXiv:1807.11205. arXiv: 1807.11205 [cs.LG].

[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *arXiv e-prints*, arXiv:1704.04760 (Apr. 2017), arXiv:1704.04760. arXiv: 1704.04760 [cs.AR].

[19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. "A Study of BFLOAT16 for Deep Learning Training". In: *arXiv e-prints*, arXiv:1905.12322 (May 2019), arXiv:1905.12322. arXiv: 1905.12322 [cs.LG].

[20] 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". In: *arXiv preprint arXiv:1609.04836* (2016).

[21] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: Citeseer, 2009.

[22] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hetchman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, Yuanzhong Xu, and Zongwei Zhou. "Scale MLPerf-0.6 models on Google TPU-v3 Pods". In: *arXiv e-prints*, arXiv:1901.11150 (Jan. 2019), arXiv:1901.11150. arXiv: 1901.11150 [cs.LG].

[23] John Langford, Alexander J Smola, and Martin Zinkevich. "Slow learners are fast". In: *Proceedings of the 22nd International Conference on Neural Information Processing Systems*. Curran Associates Inc. 2009, pp. 2331–2339.

[24] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[25] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. "ImageNet/ResNet-50 Training in 224 seconds". In: *arXiv preprint arXiv:1811.05233* (2018).

[26] *MLPerf Training v0.7 Results*. URL: https://www.mlperf.org/training-results-0-7.

[27] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv e-prints*, arXiv:1602.01783 (2016), arXiv:1602.01783. arXiv: 1602.01783 [cs.LG].

[28] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. "Massively Parallel Methods for Deep Reinforcement Learning". In: *arXiv e-prints*, arXiv:1507.04296 (2015), arXiv:1507.04296. arXiv: 1507.04296 [cs.LG].

[29] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. "Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: (2011).

[30] 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*. 2011, pp. 693–701.

[31] Steffen Rendle, Li Zhang, and Yehuda Koren. *On the Difficulty of Evaluating Baselines: A Study on Recommender Systems*. 2019. arXiv: 1905.01395 [cs.IR].

[32] 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: *arXiv e-prints*, arXiv:1409.0575 (Sept. 2014), arXiv:1409.0575. arXiv: 1409.0575 [cs.CV].

[33] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". In: *arXiv preprint arXiv:1712.01815* (2017).

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9.

[35] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *arXiv e-prints*, arXiv:1905.11946 (May 2019), arXiv:1905.11946. arXiv: 1905.11946 [cs.LG].

[36] *Training EfficientNet on Cloud TPU*. https://cloud.google.com/tpu/docs/tutorials/efficientnet.

[37] *Using TPUEstimator API on Cloud TPU*. URL: https://cloud.google.com/tpu/docs/using-estimator-api.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[39] Arissa Wongpanich, Hieu Pham, James Demmel, Mingxing Tan, Quoc Le, Yang You, and Sameer Kumar. *Training EfficientNets at Supercomputer Scale: 83Top-1 Accuracy in One Hour*. 2020. arXiv: 2011.00071 [cs.LG].

[40] Arissa Wongpanich, Yang You, and James Demmel. "Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning". In: HPCAsia2020 (2020), 52–60. DOI: 10.1145/3368474.3368498. URL: https://doi.org/10.1145/3368474.3368498.

[41] *XLA: Google's Accelerated Linear Algebra library*. URL: https://www.tensorflow.org/xla.

[42] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds". In: *arXiv preprint arXiv:1903.12650* (2019).

[43] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. "Image Classification at Supercomputer Scale". In: *arXiv e-prints*, arXiv:1811.06992 (Nov. 2018), arXiv:1811.06992. arXiv: 1811.06992 [cs.LG].

[44] Yang You, Aydın Buluç, and James Demmel. "Scaling Deep Learning on GPU and Knights Landing Clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, p. 9.

[45] Yang You, Igor Gitman, and Boris Ginsburg. "Large Batch Training of Convolutional Networks". In: *arXiv e-prints*, arXiv:1708.03888 (2017), arXiv:1708.03888. arXiv: 1708.03888 [cs.CV].

[46] Yang You, Igor Gitman, and Boris Ginsburg. "Scaling SGD Batch Size to 32k for ImageNet Training". In: *arXiv preprint arXiv:1708.03888* (2017).

[47] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes". In: *arXiv e-prints*, arXiv:1904.00962 (Apr. 2019), arXiv:1904.00962. arXiv: 1904.00962 [cs.LG].

[48] Yang You, Zhao Zhang, C Hsieh, James Demmel, and Kurt Keutzer. "ImageNet training in minutes". In: *CoRR, abs/1709.05011* (2017).

[49] Sixin Zhang, Anna E Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD". In: *Advances in Neural Information Processing Systems*. 2015, pp. 685–693.