

Distributed Training Large-Scale Deep Architectures

Shang-Xuan Zou, Chun-Yen Chen, Jui-Lin Wu, Chun-Nan Chou,
Chia-Chin Tsao, Kuan-Chieh Tung, Ting-Wei Lin, Cheng-Lung Sung,
and Edward Y. Chang^(✉)

HTC Research, Taipei, Taiwan
edward_chang@htc.com

Abstract. Scale of data and scale of computation infrastructures together enable the current deep learning renaissance. However, training large-scale deep architectures demands both algorithmic improvement and careful system configuration. In this paper, we focus on employing the system approach to speed up large-scale training. Taking both the algorithmic and system aspects into consideration, we develop a procedure for setting mini-batch size and choosing computation algorithms. We also derive lemmas for determining the quantity of key components such as the number of GPUs and parameter servers. Experiments and examples show that these guidelines help effectively speed up large-scale deep learning training.

Keywords: Deep learning · Neural network · Convolutional neural networks · Distributed learning · Speedup · Performance tuning

1 Introduction

In the last five years, neural networks and deep architectures have been proven very effective in application areas such as computer vision, speech recognition, and machine translation. The convincing factor that makes deep learning shine is *scale*, in both data volume and computation resources. Large network and large scale of training data demands scalable computation. However, scaling up computation is not merely throwing in an infinite number of CPUs and GPUs. As Amdahl's law [2] states, the non-parallelizable portion of a computation task may cap computation speedup. Non-parallelizable overheads in deep learning frameworks should be carefully mitigated to speed up training process.

Several open-source projects (e.g., Caffe [25], MXNet [7], TensorFlow [1], and Torch [9]) have been devoted to speeding up training deep networks. They can be summarized into two approaches: deep-learning algorithm optimization and algorithm parallelization. The former includes using faster convolution algorithms, improving stochastic gradient decent with faster methods, employing compression/quantization, and tuning the learning rate with advanced optimization techniques. Indeed, most open-source libraries have quickly adopted available state-of-the-art optimizations. However, most users in academia and industry do not

know how to set parameters, algorithmic and system, to conduct cost-effective training. Researchers and professionals face at least the following questions in three levels, which are intra-GPU, inter-GPU, and inter-machine:

1. With X amount of data, what is the size of each mini-batch (X_{mini}) and how to maximize GPU utilization?
2. How many GPUs (G) should be employed, and how should such a system be configured?
3. How many parameter servers (N_{ps}) should be deployed when building a distributed system?

In this work, we identify computation bottlenecks of representative frameworks and aim to answer the above questions by providing system configuration guidelines given the characteristics of the training data and hardware parameters.

1.1 Related Work

Since deep-learning training is time-consuming, many previous studies devoted to improve the training performance. These prior contributions can be divided into two approaches: algorithmic and system. The algorithmic approach accelerates the training algorithm, whereas the system approach focuses on employing improved resources to achieve parallel training. To ensure scalability, the system approach may require enhancing the training algorithm to take full advantage of the increased resources.

Algorithmic Approach. Stochastic gradient descent (SGD) is the de facto optimization algorithm for training a deep architecture. Many SGD techniques have been developed for achieving faster convergence to the global minimum. The settings of hyper-parameters such as learning rate and mini-batch size are crucial to the training performance. Hinton and Bengio [4, 21] provide recommendations on setting hyper-parameters commonly used in gradient-based training. Batch renormalization can be an effective strategy to train a network with small or non-i.i.d mini-batches [23].

More efficient algorithms can improve speed. Some FFT-based convolution schemes were proposed [31] to achieve speedup. Additionally, Firas et al. proposed three matrix layout schemes using lowering operations [19]. *Caffe con Troll* implemented a CPU-GPU hybrid system that contains several lowering operations, and at the same time, employs a simple automatic optimizer to select the best lowering. Some compression algorithms [15] were developed for both good compression ratios and fast decompression speed to enable block-wise uncompressed operations, such as matrix multiplication are executed directly on the compressed representations.

System Approach. Convolution and matrix multiplication are two common arithmetic operations used in a deep learning computation task. A GPU is well-suited for speeding up such operations since these operations are parallelizable. To achieve further speedup, the next logical step is to employ multiple GPUs, and to configure a distributed clusters of CPUs and GPUs. The computation time can be largely reduced via data parallelism and/or model parallelism. Many projects have proven parallelism to be helpful [8, 13, 22, 26, 34, 38].

According to Amdahl’s law, the peak performance of a parallel architecture is capped by the overhead portion of the computation task. In the context of deep learning, its training overhead includes synchronization between distributed threads, disk I/O, communication I/O, and memory access. To reduce synchronization delay, Zinkevich et al. [40] proposed an asynchronous distributed SGD algorithm to guarantee parallel acceleration without tight latency constraints. Chen et al. [6] proposed adding backup workers in synchronous SGD algorithm to mitigate the bottleneck. To reduce the impact of I/O on the overall speedup, most open-source frameworks attempt to conceal I/O behind computation via the pipeline approach proposed in [30]. Such approach requires a computation unit to be sufficiently long so as to hide I/O overheads as much as possible. The pipeline approach, however, demands carefully setting up the unit size of computation (or mini-batch size) and the number of parameter servers. We will propose how to best estimate these configuration parameters in Sect. 3.

Computation Frameworks. There have been several deep learning open-source efforts. Representative frameworks are CNTK [12], Theano [24], Caffe [25], MXNet [7], TensorFlow [1], and Torch [9]. Among these frameworks, MXNet and TensorFlow are built-in distributed training frameworks. Users can easily develop algorithms running on computing clusters with thousands of CPUs or GPUs. Several works are proposed to give users a glimpse on the factors that they must take into consideration. Bahrapour et al. [3] provided a comparative study on different frameworks with respect to extensibility, hardware utilization, and performance. Shi et al. [35] conducted studies on performance of selected frameworks. These works offer practitioners a high-level guideline to select an appropriate framework. Given a selected framework, our work aims to provide further configuration guidelines to make training both fast and cost-effective.

1.2 Contribution Summary

In summary, this work makes the following contributions:

1. Identifying computation bottlenecks and devising their remedies.
2. Quantifying remedies into an optimization model. We formulate our remedies into an optimization model to determine the optimal mini-batch size and carefully balance memory and speed trade-offs so as to employ the fastest algorithms given the memory constraint.
3. Recommending distributed configuration involving multiple GPUs and parameter servers.

2 Training Process and Setup

Figure 1 depicts a general architecture of deep-learning training and data flow. A worker is basically a commodity computer equipped with G GPUs. When aiming to improve parallelism via a distributed architecture, a worker and a parameter server can be replicated into multiple copies connected by a network. The training samples are divided into mini-batches. The mini-batch processing pipeline in the training process consists of seven steps. After the model parameters W and the data processing pipeline is initialized, the training process repeats until an approximate minimum is obtained.

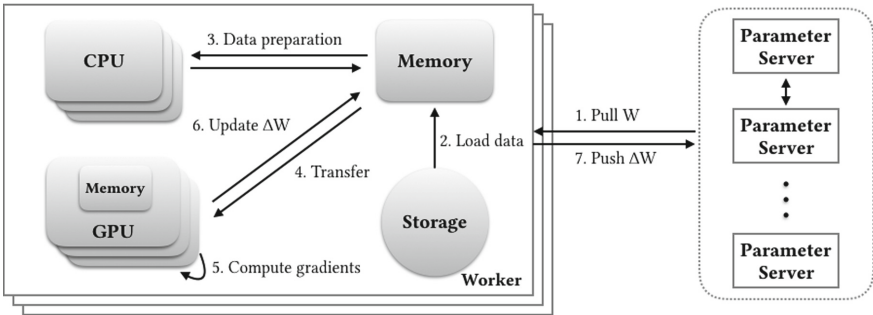


Fig. 1. Deep learning system architecture. The batch processing pipeline in the general training process can be divided into seven steps.

1. *Parameter refresh.* In distributed training, the latest copy of model parameters W is pulled from parameter servers at the beginning of each mini-batch processing. W is then loaded onto GPU memory. A distributed environment consists of N_w workers and N_{ps} parameter servers for managing shared parameters.
2. *Data loading.* A subset of the X training instances called *mini-batch* of size X_{mini} is loaded from the persistent storage to the main memory.
3. *Data preparation.* X_{mini} instances are transformed into the required input format. These instances may be augmented to mitigate the over-fitting problem and enrich sample diversity.
4. *Host to GPU transfer.* The mini-batch is loaded onto the memory of a GPU. If G GPUs are employed, G different mini-batches are loaded onto G GPUs.
5. *GPU processing.* Required computations including matrix multiplication and convolution are performed on G GPUs for the gradients against the given mini-batch.
6. *Parameter update.* The delta ΔW is derived from the gradients and applied to the previous version of W in main or GPU memory.
7. *Distributed update.* The parameter updates are sent to parameter servers when distributed machines are configured.

Among the seven steps, step 5 performs computation, and the other steps that cannot be hidden behind step 5 are considered as overheads. The larger fraction of the time which those overhead steps take, the less effective parallelism can achieve. Therefore, our tasks are minimizing overhead time and hiding overheads via pipelining as much as possible. The remainder of this paper is to demonstrate how the following parameters can be carefully tuned to achieve such goals, organized into three sections. In Sect. 3.1, we provide a procedure to recommend a mini-batch size that leads to maximum training performance. Section 3.2 provides an in-depth analysis on training in a multi-GPU environment. We provide a lemma to estimate the number of GPUs G for a desired factor of speedup. In Sect. 3.3, we address issues involving distributed workers. The communication between training hosts and parameter servers is an overhead that could seriously degrade training speedup. We propose a scheme to estimate the number of parameter servers N_{ps} , whose network capacity is B_{ps} .

We set up our evaluation environment with Elastic Compute Cloud (EC2) of Amazon Web Services (AWS)¹. All experiments run on EC2 P2 instances equipped with NVIDIA Tesla K80 Accelerators which contain a pair of NVIDIA GK210 GPUs. Each GPU provides 12 GB memory and 2,496 parallel processing cores. The CPU is a customized version of Intel Broadwell processor running at 2.7 GHz. To avoid unexpected GPU clock rate adjustment in our experiments, we disable GPU autoboot function.

We perform experiments and demonstrate our ideas with MXNet and TensorFlow. Virtual machines are launched from Amazon deep learning AMI (Amazon Machine Image) *v2.1* preloaded with NVIDIA CUDA toolkit *v7.5* and cuDNN *v5.1*. We conduct experiments on the ILSVRC-2012 dataset, the subset of ImageNet [14] containing 1,000 categories and 1.2 million images on SSD. The other set containing 50,000 labeled images is used as validation data.

3 Configuration of High Performance Training System

We study configurations in three incremental steps, starting from a single GPU (Sect. 3.1), then expanding our benchmarking to multiple GPUs (Sect. 3.2), and finally to distributed nodes where each node consists of multi-GPUs (Sect. 3.3). Each of these three steps focuses on analyzing one system configuration.

3.1 Training on Single GPU Instance

In this section, we first point out the common performance pitfalls in developing neural networks. We illustrate that the setting of mini-batch size is the primary factor that determines training speed. We then formulate selecting the mini-batch size X_{mini} as an optimization problem and provide a procedure to solve for X_{mini} that can achieve fastest training speed.

¹ GPU instances on Google Compute Engine (GCE) do not support GPU peer-to-peer access, and hence we will defer our GCE experiments till such support is available.

Identifying System Issues. Most neural networks are initially designed according to some heuristics. Researchers may not have the full picture about their model’s feasibility, convergence quality, and prediction quality unless they conducted some experiments. During the experimental process, various hyper-parameter values may be tested exhaustively by a trial-and-error process. According to our own experience, it is typically unknown at the beginning to know how long it would take to run a round of training job, let alone configure a cost-effective system that can maximize training speed. A suboptimal system configuration can lead to excessive execution time because of encountering the following issues:

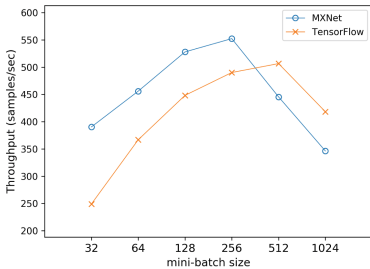
- *Shortage of GPU memory space.* A GPU cannot start computation without the data and metadata being loaded into GPU memory. A neural network designed without system knowledge may require more memory capacity than available memory. This excessive memory use may cause unnecessary thrashing and prolong training time.
- *Ineffective trade-off between speed and memory.* Deep learning frameworks may execute operations of a training task in different algorithms, which have different speed and memory-use trade-offs. The selection of using which algorithm is a layer-dependent decision. The selection factors include input data size, layer parameters, mini-batch size, and available GPU memory space. Consider the convolution operation as an example. An FFT-based implementation runs faster than a GEMM-based one but it requires more memory. The training speed may be degraded when a large X_{mini} exhausts memory capacity in order to run a faster FFT-based algorithm. Thus, when tuning factors mentioned above, we should consider the impact on memory consumption because the memory budget affects the selection of algorithm.

Selecting a good mini-batch size, one must examine from both the algorithmic and system aspects. From the algorithmic aspect, the mini-batch size is suggested to be larger than the number of output classes and a mini-batch contains at least one sample from each class [21]. The diversified training data leads to more stable convergence. From the system aspect, a proper mini-batch size helps to improve the parallelism inside GPU and enables the faster implementation of an operator. Based on the suggested mini-batch size considering the algorithmic aspect, we introduce the system aspect into deciding X_{mini} .

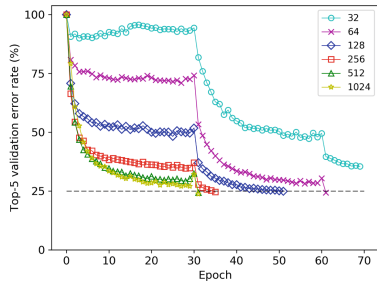
Choosing Convolution Algorithms. Speeding up convolution involves GPU memory and computation speed trade-off. There are different algorithms for implementing the convolution operation. GEMM-based implementations converts convolution to a matrix multiplication, which can be slow but the up side is that it requires less memory space. FFT-based implementations run faster than the GEMM-based by using efficient matrix multiplication and reducing the number of floating point operations. However, FFT-based implementations demand substantially more memory as the filters are padded to be the same

size as the input. In addition, FFT-based implementations require extra memory space for feature mapping on domain transformation. Take AlexNet as an example, the memory space required by the first layer with FFT is 11.6 times of that with GEMM given mini-batch size 128.

To further understand the impact of X_{mini} , we experimented with MXNet and TensorFlow, and plot system throughput (y -axis) versus X_{mini} (x -axis) in Fig. 2(a). Although different frameworks may yield different throughputs, the trend remains the same, that is, the system throughput degrades once after X_{mini} reaches a threshold. The reason why the throughput drops is that MXNet and TensorFlow run a slower version of convolution due to the constrained free memory caused by the increased X_{mini} . How to determine the optimal X_{mini} ? We next formulate the problem of determining X_{mini} as an optimization problem.



(a) Throughputs vs. mini-batch sizes.



(b) Learning curves vs. mini-batch sizes.

Fig. 2. Dual impact of mini-batch size

Optimizing Mini-batch Size. In order to formulate the problem of determining X_{mini} , we first define a memory constraint M_{bound} , which is built into the later optimization formulas for X_{mini} . During our formulation, most of the symbols follow in the same fashion of [11].

Deriving M_{bound} .

We assume that a CNN such as AlexNet [27] consists of two major components: feature extraction and classification. Further, we assume that the feature extraction part comprises of n layers where stacked convolution layers are optionally followed by pooling layers, and the classification part consists of m fully-connected layers. We use $B_i \times H_i \times D_i$ and $B_{i+1} \times H_{i+1} \times D_{i+1}$ where $i \in \{0, 1, \dots, n\}$ to represent the sizes of inputs and outputs of convolution layers (or pooling layers), respectively. In particular, the size $B_0 \times H_0 \times D_0$ represents the size of input data. If we take training AlexNet on the ImageNet [14] as the example, $B_0 \times H_0 \times D_0$ is equal to $224 \times 224 \times 3$. For the i^{th} layer of convolution and pooling layers, we denote its spatial extent (i.e. the size of filters) as F_i , its stride as S_i , its amount of padding as P_i , and its number of filters as K_i . Please note that if the i^{th} layer is a pooling layer, its K_i is equal to zero, i.e.

$K_i = 0$. Thus, the inputs and outputs in the feature extraction part have the following relations:

$$\begin{aligned}
 B_{i+1} &= (B_i - F_{i+1} + 2P_{i+1})/S_{i+1} + 1, \\
 H_{i+1} &= (H_i - F_{i+1} + 2P_{i+1})/S_{i+1} + 1, \text{ and} \\
 D_{i+1} &= \begin{cases} K_{i+1}, & \text{if } (i+1)^{\text{th}} \text{ layer is convolution layer} \\ D_i, & \text{if } (i+1)^{\text{th}} \text{ layer is pooling layer} \end{cases}.
 \end{aligned} \tag{1}$$

The memory allocated for the feature extraction part of CNNs includes the input data, outputs (i.e. feature maps) of all the layers, model parameters, and gradients. We assume that all the values are stored by using single precision floating point (32 bits). Based on the aforementioned notations and Eq. (1), the memory usage for the input data and outputs of all layers in the feature extraction part can be calculated as follows:

$$M_{FM} = \sum_{i=0}^n B_i \times H_i \times D_i \times X_{mini} \times 32. \tag{2}$$

Regarding the model parameters, there are two kinds of parameters: weights and biases. Though the biases are often omitted for simplicity in the literature, we take them into account here in order to estimate the memory usage precisely. Besides, we assume that the size of the gradients is twice as the size of the model parameters². Thus, we can derive the memory usage for the model parameters and their related gradients by the following equation:

$$\begin{aligned}
 M_{MP} &= \sum_{i=1}^n F_i \times F_i \times D_{i-1} \times K_i \times 3 \times 32 \quad (\text{weights}) \\
 &+ \sum_{i=1}^n K_i \times 3 \times 32 \quad (\text{biases}).
 \end{aligned} \tag{3}$$

Furthermore, the memory allocated for the classification part of CNNs contains the outputs of all neurons and model parameters. We use L_j where $j \in \{1, \dots, m\}$ to denote the number of neurons at j^{th} layer. Again, we make the same assumption that the size of the gradients is twice as the size of the model parameters. Therefore, the memory usage for the classification part of CNNs is as follows:

$$\begin{aligned}
 M_C &= \sum_{j=1}^m L_j \times 32 \quad (\text{outputs}) \\
 &+ \sum_{j=1}^{m-1} L_j \times L_{j+1} \times 3 \times 32 \quad (\text{weights}) \\
 &+ (m-1) \times 3 \times 32 \quad (\text{biases}).
 \end{aligned} \tag{4}$$

² For each training instance, we need to store the gradients of all model parameters. The aggregated gradients of all model parameters are also required for a specific batch.

According to Eqs. (2) to (4), the memory constraint M_{bound} can be approximately determined by the following equation:

$$M_{bound} = M_{GPU} - M_{FM} - M_{MP} - M_C, \quad (5)$$

where M_{GPU} is the total memory of a GPU in terms of bits.

Deriving X_{mini} .

Assuming that there are p kinds of convolution algorithms, and q layers in the CNN. (In the case that we have illustrated so far, $p = 2$. Other choices of convolution algorithms can be Winograd minimal convolution algorithm [28], Strassen algorithm [10], fbfft [37], etc.) The parameter $x_{k,l} \in \{0, 1\}$ represents whether the k^{th} layer uses the l^{th} convolution algorithm or not. When $x_{k,l}$ is evaluated to 1, it means that the k^{th} layer uses the l^{th} algorithm to compute convolution. The value $T_{k,l}$ is the time consumption at the k^{th} layer for the l^{th} algorithm. The value $M_{k,l}$ is the memory consumption at the k^{th} layer for the l^{th} algorithm. Thus, the problem of determining X_{mini} can be formulated an optimization problem as follows:

$$\begin{aligned} \min \quad & \sum_{k=1}^q \sum_{l=1}^p x_{k,l} \times T_{k,l} \\ \text{s.t.} \quad & \sum_{k=1}^q \sum_{l=1}^p x_{k,l} \times M_{k,l} \leq M_{bound} \quad \& \quad \forall k \sum_{l=1}^p x_{k,l} = 1, \end{aligned} \quad (6)$$

where the M_{bound} is derived from Equation (mem:bound).

Obviously, Eq. (6) is an integer linear programming (ILP) problem [32], which is NP-hard. However, there are several off-the-shelf heuristic methods and libraries (e.g. GLPK [17]) for solving ILP problems. Given a range of mini-batch sizes that can attain good accuracy, we can derive the estimated training time for each mini-batch size by solving Eq. (6). The mini-batch size which leads to the minimal training time is then the suggested X_{mini} .

This far, we assume that a CNN model is given to determine X_{mini} and layer-dependent convolution algorithms to maximize training speed. We can make two further adjustments:

- *Permit X_{mini} reduction.* The researchers may need to compromise on smaller mini-batch size if the target one is not feasible or does not deliver acceptable performance under the constraint of GPU memory size. Ghadimi et al. [16] shows that the convergence rate of SGD on a non-convex function is bounded by $O(1/\sqrt{K})$, where K is the number of samples seen, i.e., mini-batch size. It can be interpreted that a range of mini-batch sizes can deliver similar convergence quality. In Fig. 2(b), the x -axis depicts the epoch number and the y -axis depicts the top-5 validation error rate³. The figure shows that indeed

³ AlexNet achieved 18.2% top-5 error rate in in the ILSVRC-2012 competition, whereas we obtained 21% in our experiments. This is because we did not perform all the tricks for data augmentation and fine-tuning. We choose 25% as the termination criterion to demonstrate convergence behavior when mini-batch sizes are different.

a range of mini-batch sizes enjoy similar convergence quality. Therefore, we could reduce X_{mini} to increase M_{bound} to permit more memory space to run a faster convolution execution to achieve overall speedup.

- *Permit model adjustment.* Suppose that the constrained space of memory prevents us from running a faster algorithm. We could adjust the CNN model to free up some memory. For instance, if the i^{th} layer can be sped up ten times and the j^{th} only twice. To accommodate running a faster algorithm for the i^{th} layer, we could adjust both layers to e.g., use a larger stride or memory-efficient filters.

3.2 Scale with Multiple GPUs

When one GPU cannot handle the training task timely, employing multiple GPUs is the next logical step to share the workload and achieve speedup. When G GPUs are used and the maximal 100% efficiency is achieved, the speedup is G times. Let α denote the system efficiency between 0% and 100%. Lemma 1 provides the estimated efficiency given G GPUs.

Lemma 1. *Let T denote the total training time, where T can be divided into computation time T_C and overhead T_O . Let R_O denote the ratio of overhead or $R_O = T_O/T_C$. Suppose the desired efficiency of the system is α , where $\alpha \leq 100\%$. The efficiency can be estimated as*

$$\alpha = \frac{1 + R_O}{1 + GR_O}.$$

Proof. Details of the proof is documented in the extended version of this paper [41].

Lemma 1 can be used to estimate system efficiency given R_O and G , and also can be used to estimate the acceptable R_O given α and G . For example, given four GPUs and target efficiency $\alpha = 80\%$, the ratio of overhead that cannot be hidden behind computation must not exceed 9%.

To estimate R_O , a practitioner can quickly profile the training program for a couple of epochs. Some frameworks such as MXNet and TensorFlow provide the capability to visualize the execution of a training task, which can be used to derive R_O . If a computation framework is not equipped with a profiling tool, one can visualize program execution using **nvprof**⁴. Suppose a practitioner is asked to make $3x$ speedup of a training task, and she measures $R_O = 10\%$. According to the lemma, she can configure a 4 GPU system to achieve the performance objective.

To evaluate Lemma 1, we conduct the training on four neural networks to compare the estimated speedup with actual speedup. Though the estimated R_O is a constant and in real-time overheads could be stochastic, Fig. 3 shows that in all cases the estimated speedup matches the the actual speedup. Therefore,

⁴ **nvprof** only profiles GPU activities, so the CPU activities cannot be analyzed.

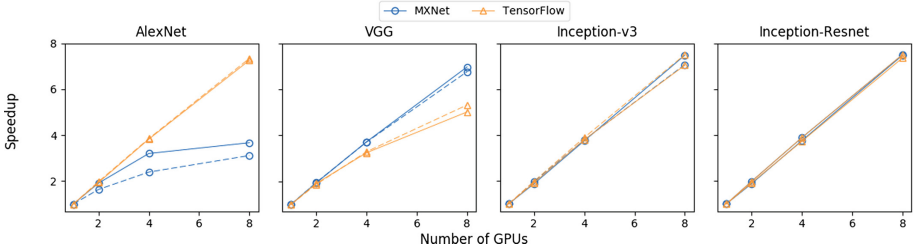


Fig. 3. Comparison of speedup (dotted-line: estimated, solid-line: actual)

the lemma can be used to estimate the performance gain of using G GPUs and devise a cost-effective training plan including system configuration and parameter settings.

The overall speedup can be improved by reducing computation overheads. We conclude this subsection by providing two overhead reduction suggestions.

- *Data transfer pipelining.* Low throughput of feeding training data is a major bottleneck that degrades the multi-GPU training performance as the demand for bus bandwidth for loading data grows with the number of GPUs. Pipelining data loading (I/O) with computation is the effective way to reduce the overhead brought by data preparation. The impact of disk I/O can be further alleviated by using better disk or reducing expensive file operations like seek. Modern frameworks such as TensorFlow and MXNet provide the way to rearrange training samples so that the data can be read in sequentially. The load for decoding and augmenting training data may cause extreme high CPU usage and drags the performance of data provision. The computation intensive jobs should be avoided on CPUs.
- *Peer-to-peer parameter updates.* Synchronizing parameter updates among GPUs, as indicated in step 6 in Fig. 1, is another common bottleneck in multi-GPU training environment. A naive implementation is to keep the latest model at main memory, transfer the latest copy to GPUs at the beginning of batch processing, and aggregate updates from all GPUs. It leads to bus contention and huge data load between main memory and GPUs under CUDA programming model. To alleviate the hot spot issue, the weight updates can be completed via GPU high-speed DMA if GPU supports peer-to-peer transfer.

If multiple GPUs with low computing overhead still cannot meet the desired performance, distributed training is the option you can consider. We'll discuss the topic in the next section.

3.3 Distributed Training

Distributed training has become increasingly important because of the growth of dataset size and model complexity. To effectively orchestrate multiple machines

for a training task, the system must provide a way to manage the globally shared model parameters. The parameter server architecture, i.e., a cluster of machines to manage parameters, is widely-used to reduce I/O latency for handling parameter updates [29,30]. As shown in Fig. 1, parameter servers maintain latest parameter values and serve all workers. The workers retrieve updated parameters from the cluster, complete computation, and then push updates back to the cluster of parameter servers.

Parameter updates can be performed either synchronously or asynchronously. Employing synchronous updates ensures consistency but suffers from the performance dragger issue. Updating parameters asynchronously gains training speed and may not significantly affect training accuracy according to prior studies [13]. When I/Os can be performed asynchronously, fetching and updating parameters can be hidden behind computation and hence computation overhead can be mitigated. We assume that an asynchronous update policy is employed.

Let N_{ps} denote the number of parameter servers. How many parameter servers should be configured to hide the computation overhead? We select N_{ps} when $N_{ps} + 1$ can no longer speed up the training task. Before we prove our lemma that derives the most effective N_{ps} , we enumerate two desired subgoals or conditions.

The first subgoal is that the computation duration of a worker should be longer than its communication time with the parameter cluster. In other words, the I/O time between a worker thread and its designated parameter servers is shorter than the computation time of that worker. This condition allows parameters being pre-fetched before a new round of computation commences. Therefore, the I/O overhead can be hidden behind computation. The second subgoal is to distribute parameter-update workload evenly among parameter servers. We assume a dynamic load-balancing policy (e.g., [5]) can be employed to distribute parameter retrieval and update workload almost evenly among N_{ps} servers.

Lemma 2. *Given a round of GPU computation time T_C on a worker, number of workers N_w , and parameter size S_p , the minimum number of parameter servers N_{ps} , whose network capacity is B_{ps} , required to mask communication I/Os is*

$$N_{ps} \simeq \left\lceil \frac{2S_p N_w}{B_{ps} T_C} \right\rceil.$$

Proof. Details of the proof is documented in the extended version of this paper [41].

Lemma 2 suggests a back-of-the-envelope estimate on N_{ps} given two ideal conditions. When the conditions do not hold, more parameter servers should be employed to be able to mask I/O overhead. Three measures are recommended:

1. *Increase T_C .* When workload cannot be evenly distributed, the computation time should be longer to mask most I/Os. Therefore, a good strategy is to maintain a large T_C . In other words, having a larger mini-batch size when the memory capacity permits is helpful. Goyal et al. [18] proposed a scheme to use

- a larger mini-batch size without loss of accuracy. Besides, a larger mini-batch leads to less number of parameter updates and improves overall performance.
2. *Improve B_{ps} .* Increasing channel bandwidth can reduce time for pushing/pulling parameters. Insufficient bandwidth of the communication channel may throttle the training performance. Thus, high speed networking is highly recommended when applying distributed training.
 3. *Balance workload.* Prior works [5, 30] propose effective data placement methods to balance dynamic workload. Such load balancing schemes can avoid I/O bottlenecks, and lead to overall overhead reduction.

4 Concluding Remarks

AlphaGo showed that more training data can only be helpful towards improving machine intelligence and competitiveness. Recently, Residual Neural Networks [20, 36] shows that in both theory and practice, more layers of neural networks correlates to a higher achieved accuracy by a trained classifier. At a 2016 machine learning workshop [33], Andrew Ng presented that the traditional biases and variance trade-off have not appeared in training large-scale deep architectures. In other words, the larger the scale, the better suited the architecture is for improving the intelligence of a “machine”.

This “larger the better” conjecture certainly demands that database and machine learning communities devise data management and data mining systems that can handle an ever increasing workload. We foresee that not only will algorithmic research continue flourishing, but system research and development will as well. Already we have seen that GPU vendors are enhancing distributed GPU implementations. Advances in interconnected technology and implementation will help reduce both I/O overhead in data loading and in parameter updates.

In this work, we provided practical guidelines to facilitate practitioners the configuration of a system to speed up training performance. Our future work will focus on effectively managing such large-scale training systems to achieve both high accuracy and cost-effectiveness in three specific areas:

- *Flexibility.* Prior work [39] provided a flexibility to work with any compatible open-source frameworks. For example, we expect to simultaneously work with multiple frameworks such as MXNet and TensorFlow to complete a large-scale training task running on Azure, AWS, GCE, and other available commercial clouds.
- *Scalability and elasticity.* In addition to the parameter estimation performed in this work, we will research dynamic schemes to adjust allocation and scheduling parameters according to the dynamic workload nature of distributed systems.
- *Ease of management.* We plan to devise tools with the good user experience for monitoring and managing the training system.

References

1. Abadi, M. et al.: TensorFlow: large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org (2015).
2. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, 18–20 April 1967, pp. 483–485. ACM (1967)
3. Bahrampour, S. et al.: Comparative study of deep learning software frameworks. In: [arXiv.org](http://arxiv.org). [arxiv: 1511.06435v3](https://arxiv.org/abs/1511.06435v3) [cs.LG], November 2015
4. Bengio, Y.: Practical recommendations for gradient-based training of deep architectures. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the Trade. LNCS, vol. 7700, pp. 437–478. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-35289-8_26](https://doi.org/10.1007/978-3-642-35289-8_26)
5. Chang, E., Garcia-Molina, H., Li, C.: 2D BubbleUp: managing parallel disks for media servers. Technical report, Stanford InfoLab (1998)
6. Chen, J. et al.: Revisiting distributed synchronous SGD. arXiv preprint [arXiv:1604.00981](https://arxiv.org/abs/1604.00981) (2016)
7. Chen, T. et al.: MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint [arXiv:1512.01274](https://arxiv.org/abs/1512.01274) (2015)
8. Chilimbi, T.M. et al.: Project adam: building an efficient and scalable deep learning training system. In: OSDI, vol. 14, pp. 571–582 (2014)
9. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: a matlab-like environment for machine learning. In: EPFL-CONF-192376 (2011)
10. Cong, J., Xiao, B.: Minimizing computation in convolutional neural networks. In: Wermter, S., Weber, C., Duch, W., Honkela, T., Koprinkova-Hristova, P., Magg, S., Palm, G., Villa, A.E.P. (eds.) ICANN 2014. LNCS, vol. 8681, pp. 281–290. Springer, Cham (2014). doi:[10.1007/978-3-319-11179-7_36](https://doi.org/10.1007/978-3-319-11179-7_36)
11. CS231n Convolutional neural network for visual recognition (2017). <http://cs231n.github.io/>
12. Dally, W.J.: CNTK: an embedded language for circuit description. Department of Computer Science, California Institute of Technology, Display File
13. Dean, J. et al.: Large scale distributed deep networks, pp. 1223–1231 (2012)
14. Deng, J. et al.: ImageNet: a large-scale hierarchical image database. In: CVPR 2009 (2009)
15. Elgohary, A., et al.: Compressed linear algebra for large-scale machine learning. Proc. VLDB Endow. **9**(12), 960–971 (2016)
16. Ghadimi, S., Lan, G.: Stochastic first-and zeroth-order methods for nonconvex stochastic programming. SIAM J. Optim. **23**(4), 2341–2368 (2013)
17. GNU linear programming kit (2012). <https://www.gnu.org/software/glpk/>
18. Goyal, P. et al.: Accurate, large Minibatch SGD: training ImageNet in 1 h. arXiv preprint [arXiv:1706.02677](https://arxiv.org/abs/1706.02677) (2017)
19. Hadjis, S., et al.: Caffe con troll: shallow ideas to speed up deep learning, April 2015. [arXiv.org](https://arxiv.org/abs/1504.04343v2). [arXiv: 1504.04343v2](https://arxiv.org/abs/1504.04343v2) [cs.LG]
20. He, K. et al.: Deep residual learning for image recognition, pp. 770–778 (2016)
21. Hinton, G.: A practical guide to training restricted Boltzmann machines. Momentum **9**(1), 926 (2010)
22. Iandola, F.N. et al.: FireCaffe - near-linear acceleration of deep neural network training on compute clusters. In: CVPR, pp. 2592–2600 (2016)
23. Ioffe, S.: Batch renormalization: towards reducing Minibatch dependence in batch-normalized models, February 2017. [arXiv.org](https://arxiv.org/abs/1702.03275v1). [arXiv: 1702.03275v1](https://arxiv.org/abs/1702.03275v1) [cs.LG]

24. Bergstra, J. et al.: Theano: a CPU and GPU math expression compiler (2010)
25. Jia, Y. et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678 (2014)
26. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint [arXiv:1404.5997](https://arxiv.org/abs/1404.5997) (2014)
27. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Pereira, F. et al. (eds.) Advances in Neural Information Processing Systems, vol. 25, pp. 1097–1105. Curran Associates Inc. (2012)
28. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4013–4021 (2016)
29. Li, M. et al.: Scaling distributed machine learning with the parameter server. In: OSDI (2014)
30. Liu, Z. et al.: PLDA+: parallel latent Dirichlet allocation with data placement and pipeline processing. ACM Trans. Intell. Syst. Technol. **2**(3), 26:1–26:18 (2011). ISSN, pp. 2157–6904, doi:[10.1145/1961189.1961198](https://doi.org/10.1145/1961189.1961198). <http://doi.acm.org/10.1145/1961189.1961198>
31. Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through FFTs. In: CoRR abs/1312.5851 cs.CV (2013)
32. Nemhauser, G.L., Wolsey, L.A.: Integer programming and combinatorial optimization. In: Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.S. (eds.) Constraint Classification for Mixed Integer Programming Formulations. Wiley, Chichester (1992). COAL Bull. **20**, 8–12 (1988)
33. Ng, A.Y.: The nuts and bolts of machine learning. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning (2016)
34. Niu, F. et al.: A lock-free approach to parallelizing stochastic gradient descent. arXiv preprint [arXiv:1106.5730](https://arxiv.org/abs/1106.5730) (2011)
35. Shi, S. et al.: Benchmarking state-of-the-art deep learning software tools, August 2016. [arXiv.org. arXiv:1608.07249v5](https://arxiv.org/abs/1608.07249v5) [cs.DC]
36. Szegedy, C., Ioffe, S., Vanhoucke, V.: Inception-v4, Inception-ResNet and the impact of residual connections on learning. In: CoRR abs/1602.07261 (2016). <http://arxiv.org/abs/1602.07261>
37. Vasilache, N. et al.: Fast convolutional nets with fbfft: a GPU performance evaluation. arXiv preprint [arXiv:1412.7580](https://arxiv.org/abs/1412.7580) (2014)
38. Zhang, H. et al.: Poseidon: a system architecture for efficient GPU-based deep learning on multiple machines. arXiv preprint [arXiv:1512.06216](https://arxiv.org/abs/1512.06216) (2015)
39. Zheng, Z. et al.: SpeedO: parallelizing stochastic gradient descent for deep convolutional neural network. In: NIPS Workshop on Learning Systems (2015)
40. Zinkevich, M. et al.: Parallelized stochastic gradient descent, pp. 2595–2603 (2010)
41. Zou, S.-X. et al.: Distributed training large-scale deep architectures. HTC technical report (2017). <https://research.htc.com/publications-and-talks>