Nexus: Bringing Efficient and Scalable Training to Deep Learning Frameworks

Yandong Wang, Li Zhang, Yufei Ren, Wei Zhang
IBM Thomas J. Watson Research Center, New York, USA

Abstract—Demand is mounting in the industry for scalable GPU-based deep learning systems. Unfortunately, existing training applications built atop popular deep learning frameworks, including Caffe, Theano, and Torch, etc., are incapable of conducting distributed GPU training over large-scale clusters.

To remedy such a situation, this paper presents Nexus, a platform that allows existing deep learning frameworks to easily scale out to multiple machines without sacrificing model accuracy. Nexus leverages recently proposed distributed parameter management architecture to orchestrate distributed training by a large number of learners spread across the cluster. Through characterizing the runtime behavior of existing single-node based applications, Nexus is equipped with a suite of optimization schemes, including hierarchical and hybrid parameter aggregation, enhanced network and computation layer, and quality-guided communication adjustment, etc., to strengthen the communication channels and resource utilization. Empirical evaluations with a diverse set of deep learning applications demonstrate that Nexus is easy to integrate and can deliver efficient distributed training services to major deep learning frameworks. In addition, Nexus’s optimization schemes are highly effective to shorten the training time with targeted accuracy bounds.

I. INTRODUCTION

Deep learning has recently made stunning breakthroughs in solving complex machine learning tasks, including image classification [18], machine translations [4], and speech recognition [10], etc. Besides the advances in machine learning algorithms, such an accomplishment is greatly attributed to the continuously enhanced computing devices and the availability of big data. In other words, fast computing devices efficiently empower deep neural networks (DNN) to identify critical features from large volumes of labeled data.

Recent system approaches to satisfying the enormous computing demand for training DNNs can be generally classified into two categories, multi-CPU scaleout training and multi-GPU single system training. Multi-CPU scaleout training leverages hundreds of thousands of commodity machines to conduct collective training over the same datasets. The representative work belonging to this category include DistBelief [11] and Project Adam [7]. These systems assume that DNN models are too large to fit into any single node and CPUs are the only available computing devices. In such case a single machine cannot deliver meaningful training quality in a reasonable time. Considering the cost-efficiency issue suffered by the above approach, multi-GPU single system training uses much smaller DNN models to battle against the overfitting and using massive-parallel GPU devices to carry out the training on a single node. It has recently attracted increasing interest from both industry and academia. Lately, works [13], [4] within this category have yielded world-class results in many machine learning domains, e.g. ImageNet competition.

Recognizing the benefits of GPU-oriented approach, many deep learning frameworks have been introduced over the past few years to facilitate the design and research on deep neural networks. Among different frameworks, three notable ones are Caffe [12], Torch [3], and Theano [5], all of which use GPU as the primary computing device and target single-node based training. In addition, to achieve efficient GPU utilization, fast kernels including cuBLAS and cuDNN [6] have been widely adopted by these frameworks to speed up matrix operations, e.g. convolutions and multiplications, involved in the DNN training. While the GPU-based learning has substantially shortened the training time, the demand to further improve the training performance remains strong. A potential solution to enhance the performance is to scale out the training via leveraging distributed GPUs across multiple machines. However, achieving so requires a high-performance parameter orchestration system that is capable of effectively and efficiently hiding and reducing the communication cost. The communications include the data movement between host and device memories, and the data transfer across the networks. The expensive communication can severely damage the GPU utilization, rendering distributed training a futile effort. Due to such challenge, the aforementioned deep learning frameworks remain single-node based without distributed training support.

To conquer such a challenge, we have designed a high-performance parameter orchestration platform, named as Nexus, for existing deep learning frameworks to enable efficient and scalable DNN training. Nexus adopts data-parallel execution model to let DNNs built atop the current frameworks continue running on dis-
distributed GPUs and rely on Nexus to carry out the underlying parameter movement and aggregation. In addition, Nexus has conducted a thorough optimization to alleviate the overhead caused by the distributed training. More specifically, optimizations are performed through two complementary approaches. First, via carefully examining the design details of existing deep learning frameworks and exploiting the unique properties of DNNs, Nexus introduces several software tactics, including hierarchical and hybrid parameter aggregation, intermediate parameter caching, as well as quality-guided frequency adjustment to reduce the traffic over the network. Moreover, when the parameters exhibit high sparsity, Nexus allows dynamic format conversion to further reduce the amount of network traffic. Second, Nexus strives to exploit the full potential of high-performance hardware available in the cluster to accelerate parameter movement and aggregation. Concrete schemes to achieve this objective include leveraging RDMA protocol for transferring matrices of large sizes and using GPUs to conduct parameter aggregation.

To evaluate the performance, we have fully integrated Nexus with Caffe, Torch, and Theano. We employed 6 distinct, popular DNN models written by these frameworks to scale out their training. The experimental results confirm the efficacy and efficiency of Nexus, showing near linear scalability in terms of the input processing throughput for 5 of the 6 models compared to the original single-node cases. Such enhanced training throughput directly leads to substantially faster convergence speed, allowing Nexus to deliver the same accuracy with significantly shorter training time.

II. BACKGROUND

Though concrete neural networks vary among different deep learning applications, their fundamental training processes bear strong resemblance. In this section, we delineate a major neural network architecture, Convolutional Neural Network (CNN), that has been broadly adopted by many applications, and then we describe the CNN training procedure. Lastly we illustrate the mechanisms that allow neural networks to achieve parallel execution in distributed environments.

A. Deep Neural Network

A typical CNN comprises multiple convolution layers with one or more fully-connected layers residing at the last to perform final output computation. As the core component of CNN, each convolution layer aims to detect certain features from the input. It consists of a group of units (a.k.a. neurons) organized in a 3-dimension manner to simulate the biological visual cortex. Many CNN implementations also interweave convolution layers with pooling layers, e.g. max-pooling, to downsample the input volume along the forward pass. Usually, a CNN is equipped with a differentiable loss function to assess the quality of the prediction.

Unlike traditional neural networks, in which each unit within a layer always connects to all the units in the previous layer, a unit within a convolution layer only connects to a small region called receptive field, of the previous layer, as shown in Figure 1. A receptive field extends through the entire depth of the input. In the case shown in Figure 1, the depth of the input image is 3, representing 3 distinct color channels. To detect some desirable features from the field, a unit applies a filter to examine the receptive field of the previous layer. A filter is essentially a set of parameters that need to be learned during the training session. Also, to control the number of parameters, all the units within the same depth slice share the same filter. Units in different depth slices are designed to extract different features, thus learning different filters. So altogether, the receptive field size and the depth of the layer determine the number of parameters within a convolution layer (the parameter set is denoted by \( \omega_{conv} \) in the figure). While scanning the receptive fields, each unit conducts the dot product between the input and the filter, adds up the bias \( (b_{conv}) \), and then triggers an elementwise activation function, e.g. ReLU or tanh to generate the output of the layer. At the end, to predict the classes to which an image may belong, CNN uses a fully-connected layer to generate the class label probabilities, represented by a vector of \( D_{fc} \) elements (\( D_{fc} \) is the number of classes). As its name indicates, each unit within a fully-connected layer is linked to all the units from the previous layer, and each unit also applies a learnable filter with a predefined activation function to compute the class scores.

Throughout the training process, repeated forward pass and back propagation are applied by the frameworks to refine the filters contained in different neural network layers. To enhance the computation efficiency, batch-based processing is commonly adopted to transform a batch of images into a large matrix so that highly-optimized matrix operations, e.g., `cublasSgemm` can be leveraged to accelerate forward pass and backward pass. After processing a batch of input, accumulated gradients are then clipped and normalized, then applied to the weights of the filters. Figure 1
shows an example of updating the parameters, including $w_{conv}, b_{conv}, w_{fc}, b_{fc}$, using Stochastic Gradient Descent (SGD) during the backpropagation.

### B. Distributed Neural Network Training

Shored up by numerous theoretical studies [24], [21], [16], [23], [17], data parallelism has been widely embraced by the machine learning community to achieve parallel neural network training in distributed environments. As neural networks generally manifest neither convex nor concave property with the existence of many local optima, it is ideal to have many learners to conduct exploration over different datasets simultaneously. Throughout the training, multiple replicas of the same model are trained independently over different subsets of the training data. To avoid chaotic divergence among model instances, the learners managing the instances communicate with a centralized parameter server [14], [7] regularly to obtain updated global weights, which are kept fresh by using the parallel stochastic gradient descent (SGD) algorithm. Over the past few years, many variants of the parallel SGD have been introduced, including parallelized SGD [24], downpour SGD [11], Elastic Averaging SGD [21], etc. Though they differ in details, algorithms commonly follow the steps as described below. Each learner computes local gradients during the backpropagation phase using local parameters. Periodically, the learner checks if the condition for push or pull has been satisfied (generally governed by a communication frequency threshold). If so, the pull function fetches the global weights from the server whereas the push function sends the locally accumulated gradients or local weights to the parameter server.

As the volume of the parameters within a neural network can reach beyond hundreds of millions or even billions, the parallel SGD is under severe communication constraints. To overcome such a challenge, many previous studies, e.g., Elastic Average SGD [21], have proposed to use elastic difference, i.e., $\eta \times (w_l - w_g)$ ($\eta$ is a moving rate which requires tuning) to allow each learner to explore more optimization spaces locally before fetching the global weights without violating the convergence guarantees, thus reducing the communication frequency between the learners and the servers.

In this work, we have observed that allowing each learner to accumulate local gradients for several batches before conducting the model exchange following the Bulk Synchronous Parallel (BSP) model delivers far better convergence speed than the asynchronous counterpart for GPU-based training. Thus this is used as the default aggregation method applied by Nexus.

1When a model does not fit into a single GPU, model parallelism can be leveraged to partition a model across multiple GPUs, which collectively train a single model instance, known as model parallelism.

### III. Characterizations

Before presenting the system design, we first study the runtime behavior and performance characteristics of 6 real-world deep learning models written by Caffe, Theano, and Torch, respectively. This study helps us understand the current frameworks from multiple facets, such as data layout, parameter attributes, and computation efficiency of GPU devices, so that Nexus can be effectively designed to serve these frameworks.

#### A. Methodology

Our characterizations specifically aim to answer the following questions to help design Nexus:

1) **GPU Efficiency:** How efficient are existing deep learning frameworks using GPU devices, and how much challenges fast GPU computation is posing on the system design?

2) **Parameter Layout:** How existing frameworks organize the parameters of different DNN layers during the training, and whether the current data layout can facilitate the parameter movement across the networks without incurring excessive amount of intermediate I/O?

3) **Parameter Properties:** What is the sparsity of the parameter matrices generated during the training and whether there exist optimization opportunities that can help yield efficient I/O reduction?

Table I lists the models we used to answer the above questions along with their basic characteristics, including the framework each model is built upon and the type of neural network each model belongs to. In addition, details about the number of learnable weight matrices and the total number of learnable parameters are also included with the last column showing the datasets employed to study each model. We have used 3 popular benchmark datasets to conduct the characterizations. These datasets are ImageNet-1K [18] and CIFAR-10 [2] for image classification and WMT English-French datasets [1] for statistical machine translation.

As current frameworks lack the ability to scale out, all the experiments in this section were run on a single machine. Unless otherwise stated, each machine is equipped with 2 NVIDIA Tesla K40m GPUs spread across two sockets. Each GPU has 12GB device memory with peak single precision floating point performance of 4.29 TFLOPS. Each machine also has 2 3.3GHz Intel Xeon E5-2667 Octa-core processors with 256GB memory. The machine runs Linux Redhat 2.6.32 with CUDA 7.5. However, because GPUs in the above machines cannot communicate with each other through GPU Peer-2-Peer (P2P), we run the single node multi-GPU tests on a separate machine that is equipped with 4 Tesla K80s connected through PCIe-3 bus for the sake of comprehensive performance study.
B. Observations and Implications

Observation 1: In single GPU training, with prefetching enabled, current frameworks are able to maintain high GPU SM core utilization of \(\approx 95\%\) on average throughout the training. In contrast, device memory space utilization is relatively low (consistently \(\leq 50\%\)) as most trainings prefer batches of small sizes to larger ones given the concern of convergence speed. However, small batches lead to even faster processing time and consequently lead to more frequent model exchanges between learners and the parameter servers if conventional parallelized SGD continues to be used. Thus it is highly challenging to place the model exchanges on the critical path of the training process.

Implications: Fast GPU computation essentially entails 2 implications from both system design and algorithm perspectives: (1) It is imperative to overlap the communication with the computation whenever possible. While if the communication consistently lags behind the computation, simply allowing the computation to move forward with stale weights can severely damage the training quality. Such a phenomenon has also been confirmed by previous studies [9]. (2) It is critical to leverage and explore the feasibility of parameter aggregation algorithms that require infrequent data communications, and to allow dynamic communication frequency adjustment based on different network conditions.

Observation 2: In single node multi-GPU training, hardsync-based gradient aggregation [22] adopted by the current frameworks\(^2\) scales well when the model size is small (shown by GoogleNet), but does not scale efficiently with large model (shown by AlexNet) as illustrated in Figure 2, as each learner needs to stop the computation and to communicate with other learners to normalize the gradients after each batch processing. As a result, communication cost increases with model size, causing frequent GPU stalls.

Implications: For large models, strictly following hardsync-based aggregation is unable to exploit the performance of local GPU devices. Similar to the implications from observation 1, It is critical to explore alternative aggregation mechanisms to eschew per-batch synchronization without degrading model accuracy.

Observation 3: To build neural networks, current frameworks, Caffe and Theano, group weights or gradients of the same layer in a contiguous memory area, but store parameters belonging to different layers across different memory regions. While Torch resorts to a continuous memory chunk to store all the parameters. The sizes of parameters of different layers exhibit uneven distribution with a small number of layers, such as fully-connected layers in CNN or word-embedding in NMT, dominating the parameter size of the entire network. Figure 3 illustrates this phenomenon using 2 representative neural network models written by Theano and Caffe respectively. Taking NMT as an example, the total gradient size is \(\approx 1.83GB\), but it mainly comes from 2 word-embedding matrices, \(\text{Wemb}_\text{dec}\) and \(\text{Wemb}\), which contribute \(\approx 63\%\) of the total parameter size. Similarly, the last 3 fully-connected layers dominate the model size of AlexNet. In addition, the memory areas holding the parameters remain invariant during the training.

\(\text{2Berkeley Caffe uses aggregation tree based aggregation while Torch uses allReduce based aggregation.}\)
By studying the parameter density, we find that reduced precision poses destructive impact on the training quality as demonstrated by Figure 5.

Observation 4: Attribute analysis of all the models shows that weight matrices generated during the trainings exhibit high density when measured with default 32-bit IEEE 794 float format, whereas the density of gradient matrices differs among different models and layers as revealed in Figure 4, which illustrates the gradient matrix density of the dominating layers of different models using 3 software-based epsilons to measure the floating point values. As shown in the figure, when higher precision is used ($\epsilon = 10^{-6}$), gradients within the CNN models are commonly dense matrices. In contrast, the gradient matrices of the word-embedding layers of NMT can be highly sparse as the number of words learned in each batch only modify a small portion of the entire vocabulary. In addition, for AlexNet and GoogleNet, lowering the floating point precision can significantly dilute the density of the gradient matrices. As sparse matrices offer good opportunity to reduce I/O, an interesting question arises from this observation. If we dynamically convert the gradient matrices into sparse matrices by lowering the precision, what impact will it generate on the training quality? To answer this question, we use Imagenet-1K to train AlexNet and GoogleNet. During the training, we dynamically round the precisions of the gradients when it is smaller than $\epsilon = 10^{-6}$. However, we observe that reduced precision poses destructive impact on the training quality as demonstrated by Figure 5.

Implications: By studying the parameter density, we hope to obtain sufficient knowledge about how to reduce the memory footprint, and subsequently reduce the I/O traffic over the network and PCIe bus. The above evidence shows that directly conducting compression over the parameter matrices is unlikely to provide satisfactory volume reduction as most matrices are highly dense, and using tricks, such as lowering the precision of data representation, to increase the sparsity can be detrimental to the training quality. However, as equally evidenced by the NMT-Theano case, differentiated processing shall be conducted for different models as the dominating layers of certain models can be highly sparse. Therefore, introducing a mechanism that can efficiently reduce the gradient matrix volume during the training is still valuable for reducing the network traffic.

IV. Design of Nexus

This section presents detailed design of Nexus. we first describe the system architecture and critical components. then we discuss the programming interfaces. Following that, we illustrate the optimization schemes that allow Nexus to fulfill the performance requirements and how Nexus handles abnormal situations.

A. System Architecture

Nexus runtime, as depicted in Figure 6 (a), comprises three key components, including a group of Nexus servers that collectively store and aggregate the model parameters from multiple jobs, and Nexus client library that connects each learner with the cluster, as well as a centralized managing process, called Coordinator, which is responsible for all the logistics functionalities, e.g. job configuration and runtime monitoring.

To launch a training job, the Coordinator selects a list of Nexus servers from the server pool and carries out corresponding job configuration. The same list is assigned to all the learners of the same training. Throughout the training, Nexus client adopts intra-layer data partitioning to evenly divide the entire model used by the learner based on the number of available servers, and sends partitions to different servers according to the partition ID. As all the learners of the same training follow exactly the same model partitioning scheme, the same partitions from different learners are gathered by the same server, which then conducts user-specified aggregation function to generate updated parameters.

1) Nexus Server Design: As a throughput-critical system, Nexus server leverages lockless queues at both the network and the computation layers to achieve efficient resource utilization as illustrated in Figure 6 (b). Nexus allows data movement using both TCP and RDMA protocols to further improve the network throughput. When RDMA is enabled, it is only used for transferring the model partitions with TCP continuing.
to be used for control messages. Upon detecting the arrival of a model partition, a receiving thread determines whether it is necessary to invoke the aggregation based on the triggering condition defined by the job. For example, Downpour SGD invokes the aggregation whenever a new partition is received while BSP-based model averaging waits until all partitions are gathered before triggering the computation. When a job is ready to start the aggregation phase, the aggregation scheduler enqueues the computation into either the CPU-based aggregation queue or the GPU-based queue by taking into account of both the estimated aggregation time and the potential waiting time of each queue. Upon receiving the pull requests, the response threads are invoked to send the new model partition back to the learners.

2) Nexus Client Design: Nexus client is a critical component to reduce the network traffic during the training. This is mainly achieved through the introduction of two major constituents, including local aggregator and parameter cache as shown in Figure 6 (c). As many aggregation operations exhibit commutative and associative properties as described in section II-B, local aggregation is introduced to preprocess the models from multiple local learners before issuing combined results out to the remote servers. Besides the obvious benefit of reducing the load over the network, such hierarchical aggregation further reduces the capacity and computation burden on the server sides. However, processing speeds of local learners can still differ significantly and waiting for the slowest learners can inefficiently penalize faster learners. To mitigate such an issue, parameter cache is designed to buffer the updated parameters received from the servers, when the slow learners try to exchange the model, cached parameters can be directly returned to allow computation to proceed.

In many cases, each local learner may want to leverage multiple local GPUs to accelerate the processing by dividing the batch among multiple GPUs. However, as revealed in Section III-B, single learner multi-GPU is unable to deliver efficient scalability when model size reaches beyond certain threshold. To address such an issue, Nexus client introduces a hybrid aggregation mechanism to split a single learner into multiple groups. Each group uses a subset of GPUs to carry out hardsync-based gradient aggregation and only the group leader periodically communicates with the local aggregator in the background to exchange the model weights with the parameter server. By using such mechanism, the number of GPUs involved in the hardsync is small, thus balancing the batch processing speedup with GPU underutilization. Meanwhile, periodical weights exchange prevents different groups from diverging significantly.

B. Programming Interfaces

Nexus aims to provide simple interfaces to allow different deep learning frameworks to easily support distributed training. Overall, Nexus exposes two major C interfaces, push and pull, to let a learner send out local parameters and retrieve updated models from the server sides. Each learner has the flexibility to specify where is the parameter located, either GPU memory or CPU memory or whether the functions shall immediately return by specifying several arguments within a configuration file. Underlying, Nexus abstracts the parameters belong to the same layer as a matrix. When transferring the data, Nexus does not use any parameter serialization or deserialization and directly moves all the data in binary format. The configuration set by the Coordinator before starting the training specifies the data type, either float or double. To allow frameworks written by different languages, such as Python and Lua to use Nexus, ctype-based Python interfaces and Luarocks-based functions are provided so that Nexus can directly access the memory addresses used by those frameworks for storing all the parameters.

Figure 7 illustrates the minimum amount of changes needed to allow Neural Machine Translation (NMT) application written by Theano to use Nexus for distributed training. After importing the class from the package, the application only needs to call the Nexus push and pull APIs to exchange the model parameters with all the other learners of the same training.

```python
from ps_interface import paramClientSide
psClient = paramClientSide(...)  # push(参数)
psClient.push(training_parameters)  # 计算
psClient.pull(updated_parameters)
```

Figure 7. Changes needed to let NMT application written by Theano support Nexus-based distributed training.
C. Communication Optimizations

Figure 8. Performance benefits of using sparse vector based push.

1) Exploiting Sparsity for Traffic Reduction: As characterized in Section III-B, gradient matrices generated by many deep learning applications, such as NMT, can be highly sparse, thus providing Nexus with an opportunity to reduce the load over the network. Although using compression mechanism is an intuitive approach to achieve such an objective, our evaluations with different CPU-based compression algorithms including lightweight LZO and LZMA, show that expensive compression/decompression computation can quickly diminish all the benefits received from transferring smaller datasets. On the other hand, as the non-zero elements are randomly distribute across the entire matrix space, using trick that finds the smallest bounding rectangle that covers all the non-zero items cannot deliver satisfactory size reduction. Fortunately, we find that directly converting matrix into a sparse vector through using multiple threads is a highly effective approach to realize a high compression ratio without incurring prohibitive computation cost thanks to the high memory bandwidth and prefetching mechanism. Figure 8 compares the push performance of sparse-vector based approach and compression-based tactic, as well as the straightforward push under different network bandwidth using NMT model. As shown in the figure, sparse-vector based approach can efficiently shorten the model push time compared to all the other alternatives. It is worth noting that the current Nexus depends on the hints from the applications to determine whether the above traffic reduction is enabled, as the domain experts possess better knowledge about the potential sparsity of the matrices than the Nexus runtime does.

2) Quality Guided Communication Adjustment: To further prevent the server cluster from being heavily overloaded, Nexus leverages another characteristic that many deep learning models can tolerate a lower model exchange frequency than originally specified while retaining the same convergence speed to reduce the amount of communication. Nexus achieves so through a dynamic frequency adjustment scheme that is guided by the training quality. During the training, after every $T$ times of model exchanges, the leading Nexus server for the job proposes a new communication frequency to all the learners, who then use the new value to continue the training. Meanwhile, each learner also measures the training quality under the new frequency and compares it with the quality of the older one. If evident quality loss is observed, the learner issues rejection back to the server, which then informs all the rest learners to fall back on the older frequency value. More importantly, we enforce each learner to memorize the training quality associated with all the previous frequency values so that eventually the training can roll back to use the original one, as the training quality is improved marginally at the late phase of the training. To assess the training quality, we currently use averaged loss over the past 10 test reports. Our evaluation in Section V-A confirms the efficacy of such dynamic adjustment using a communication frequency sensitive DNN model, showing nearly identical training performance as that of using aggressive communication frequency.

D. Resilience to Abnormalities

1) Fault Tolerance: Nexus tackles the fault tolerance through checkpointing. More specifically, Nexus takes advantage of the checkpoint capabilities already available from the deep learning frameworks to let each learner periodically snapshot its own learning status, but only one of the learner upload its checkpoint status to the parameter servers. Upon detecting the failure, Nexus halts the training, rollback the status to the latest checkpoint, then continue the training.

2) Straggler Mitigation: To cope with the potential performance variation among different learners, Nexus server allows user-defined aggregation function to specify if the aggregation condition can be triggered before all the parameters have been gathered from all the learners. On the other side, after the waiting for the response reaches beyond a threshold, Nexus client determines whether a partially collected model can be returned back to the learner for the next rounds of computation based on the configuration. In addition, the Coordinator continuously monitors the training progress of different learners, when confirmed as a severely straggling process, Nexus removes it from the corresponding training(s), replaces it with a newly selected learner.

V. EVALUATION

Table II

<table>
<thead>
<tr>
<th>Model</th>
<th>Batch Size</th>
<th>Comm Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>256</td>
<td>10</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>VGG (caffe)</td>
<td>128MB / 32MB</td>
<td>1</td>
</tr>
<tr>
<td>NMT</td>
<td>64</td>
<td>50</td>
</tr>
<tr>
<td>NIN (100G)</td>
<td>64</td>
<td>10</td>
</tr>
</tbody>
</table>

We evaluate the performance of Nexus by answering the following two questions. First, how well Nexus can fulfill its goal of providing scalable training for deep learning models implemented by different frameworks? Second, how effective is each individual optimization in improving the training performance?
To answer the first question, we have composed a workload that consists of 18 training jobs using the models and the datasets listed in Table I. Each training job requires different number of GPU-based learners, ranging from 2 to 8. We randomly shuffle the order of all the training jobs and submit the head job whenever there are available GPU resources. This set of experiments were running on a cluster consisting of 24 machines, featuring 48 K40m GPUs in total, and Nexus servers were deployed on the 12 machines among them. All the machines are connected through both 10 GigE and 56Gbps InfiniBand networks. The configuration of each machine is the same as described in Section III-A. All the training jobs use SGD algorithm to minimize the objective functions. Table II further lists several key configurations, including the batch sizes and communication frequency, employed by different trainings. We use BSP-based model averaging as the aggregation method. The results of this set of experiment are shown in Figure 9 and Figure 10.

To answer the second question, we used a large model available to us from NMT to stress the critical path along the communication paths. This set of experiments used a single Nexus server with different number of learners. Detailed performance dissection is provided to examine the effectiveness of different optimization schemes. The results are illustrated in Figure 12.

A. Scalable Training with Nexus

A major feat of Nexus is to allow a diverse set of deep learning models written by different frameworks to efficiently scale out to obtain accelerated convergence. Figure 9 confirms this by showing the training throughput scalability of 6 distinct models. As shown in the figure, Nexus delivers near linear scalability to the trainings when more learners are added except for the NIN Caffe-CIFAR10 case, which features short batch processing time (≈ 20 milliseconds per batch) and requires very intensive model exchange frequency to achieve desirable convergence quality. Overall, for trainings that contain 8 learners, Nexus yields up to 7.31×, 7.42×, 3.5×, 7.69×, 7.33×, and 7.2× throughput improvement for Caffe-Alexnet, Caffe-Googlenet, Caffe-NIN, Theano-NMT, Torch-VGG, and Torch-NIN, respectively. Correspondingly, for the same set of trainings, only 11.6%, 6.7%, 40.6%, 3.7%, 6.3%, and 5.3% of training time are spent on the communication, indicating low overhead imposed by Nexus.

An immediate impact brought by the enhanced throughput is accelerated training convergence speed, which is the fundamental goal of Nexus. In this paper, we present the convergence efficiency by plotting the accuracy achieved with the test datasets along the training time. Figure 10 illustrates the convergence performance of all 6 different models trained with different number of GPUs. Overall, as shown in the figures, compared to the existing DL frameworks (1 GPU cases) that cannot leverage distributed GPUs, Nexus provides substantially faster training performance, allowing current models to achieve expected level of accuracy with significantly less amount of training time.

In addition, we have made three critical observations. First, although scaling out the GPU-based training can effectively accelerate the convergence speed, as the peak accuracy and convergence speed is constrained by the models and used datasets, simply adding more learners offers diminished training quality improvement once a threshold is reached. As shown in all the subfigures, significant accuracy improvement is observed when we increase the number of learners from 1 to 4, while further increasing the number of learners from 4 to 8 brings much less convergence speedup. Taking NIN-Torch as an example shown in Figure 10 (f), to reach 30% accuracy, original Torch needs 9.25 hours, while Nexus delivers the same accuracy with 4 learners in 3.49 hours (> 2.65× speedup). However, the same accuracy is reached in 2.8 hours with 8 learners, which only offers ≈ 20% speedup compared to 4-learner case.

Second, many training jobs, including AlexNet, GoogleNet, and VGG as well as NIN, can tolerate lower model exchange frequencies than originally specified by domain experts, providing Nexus with optimization spaces to reduce communication traffic. During the trainings, Nexus adaptively lowered the frequencies for the above four models by 2 times on average without detecting significant training quality reduction. Figure 11 illustrates the effectiveness of dynamic frequency tuning using VGG-Torch as an example. As shown in the figure, Nexus delivers nearly the same convergence speed as manually tuned frequency. On the contrary, without runtime information, manually tuning down the frequency to reduce the communication cost can damage the training quality, which is evidently demonstrated by Freq-15 and Freq-20 shown in the figure.

Thirdly, no single aggregation configuration can serve all the trainings equally well and careful tuning is necessary to select the configuration based on the models and datasets. For example, although using communication
three critical paths have been identified as the major
bottlenecks along the communication path with the number
on top of each bar showing the number of learners used
to return the aggregated output, slightly longer push time is observed
in the experiments. As shown in the figure, the Nexus
on the left side using TCP as the network protocol
and CPU-only aggregation incurs quite high overhead
with the network utilization is less than 45% on the
client side. Then to reduce the cost of the aggregation,
GPU is used to accelerate the aggregation, which proves
to be highly effective, reducing the aggregation time
by up to 15.9 ×. However, model transfer remains as
bottlenecks. Thus, we change the network from 10
GigE to InfiniBand and continue using TCP as the
protocol (TCP on InfiniBand is known as IPoIB). With
faster network, parameter push and pull are substantially
accelerated, reducing the total cost by another 50% on
average for different learners. However, IPoIB suffers a
major issue that it cannot fully utilize the bandwidth
available from InfiniBand. Therefore, Nexus is then
modified to use RDMA protocol to transfer the param-
ers. As a result, the entire cost is further cut down by
30% on average, and network utilization reaches above
75%. After network utilization is improved, Nexus then
leverages local model aggregation and caching to further
minimize the network traffic. By reducing the repeated
model exchanges, local aggregation with cache further
improves the synchronization time by another 19% on
average. As the local aggregation poses an implicit
synchronization barrier before pushing the locally ag-
gregated output, slightly longer push time is observed

B. Impact of Different Optimizations

Nexus strives to leverage high-performance hardware,
including GPU devices and RDMA-capable networks,
along with software tactics to mitigate the cost invoked
by the distributed training. Throughout the evaluation,
three critical paths have been identified as the major
bottlenecks, which include parameter push from clients
to the server cluster, and parameter aggregation on the
server sides, as well as parameter pulling to return the
updated parameters from the servers back to the clients.
In this section, we investigate how different optimization
schemes used by Nexus successfully alleviate the cost
on these three parts. To carry out the experiment, we
used a 350MB NMT model. All the experiments were
run on 4 nodes, in which 1 machine was used as a
Nexus server and the rest were used as learners without
computation to inject the models into the network.
During the experiments, we used synchronous model
averaging as the parameter aggregation method.

Figure 12 dissects the time spent on different parts of the communication path.
The figure shows the breakdown of time spent on parameter push, aggregation,
and pull. The time is measured in seconds for different models written by
Caffe, Theano, and Torch, respectively. The models include AlexNet, GoogleNet,
Network-in-Network, and VGG-Torch. The figure highlights the benefits of
RDMA and IPoIB in reducing the cost of communication.

Figure 11. Effectiveness of quality guided communication adjustment on VGG-Torch using CIFAR-10 dataset.

Figure 10. Convergence efficiency of 6 different models written by Caffe, Theano, and Torch, respectively. AlexNet and GoogleNet are built atop Caffe and evaluated with ImageNet ILSVRC12 datasets. Net-in-Net Caffe is evaluated with CIFAR-10 dataset. NMT is written by Theano and assessed with WMT12 datasets. VGG-Torch and Net-in-Net Torch are trained with CIFAR-10 and ILSVRC12 respectively.
based deep learning in an non-negligible extent. We observed that using staled parameters affect GPU-studied ML applications well according to the paper, compared to the cases without local aggregation. Nexus is capable of providing scalable training with Caffe, Theano, and Torch adequately demonstrates that evaluation with a diverse set of DL models written by such as RDMA-capable networks and GPU devices efficiently utilizes different high-performance hardware, to reduce the network traffic. In addition, Nexus efficiently utilizes different high-performance hardware, such as RDMA-capable networks and GPU devices to accelerate the data movement and aggregation. Our evaluation with a diverse set of DL models written by Caffe, Theano, and Torch adequately demonstrates that Nexus is capable of providing scalable training with significantly faster convergence speed.

REFERENCES