A Performance Evaluation of Distributed Deep Learning Frameworks on CPU Clusters Using Image Classification Workloads

Andreas Krisilias, Nikodimos Provatas, Nectarios Koziris
National Technical University of Athens, Athens, Greece
{akris, nprov,nkoziris}@cslab.ece.ntua.gr

Ioannis Konstantinou
University of Thessaly, Lamia, Greece
ikons@uth.gr

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671461

***Abstract*—Over the recent years, deep learning is widely being used in a variety of different fields and applications. The constant growth of data used to train complex models, has opened research in the distributed learning. In this domain, two main architectures are used to train models in a distribution fashion, all-reduce and parameter server. Both support synchronous learning, while parameter server also supports asynchronous learning. These architectures are adopted by tech companies, which have developed multiple systems for this purpose. Among the most popular and widely used distributed deep learning systems are Google TensorFlow, Facebook PyTorch and Apache MXNet. In this paper, we quantify the performance gap between these systems and present a detailed analysis to discuss the parameters that affect their execution time. Overall, in synchronous learning setups, TensorFlow is slower compared to PyTorch by average $2.65X$, while the latter lags MXNet by average $1.38X$. Regarding asynchronous learning, MXNet is faster by average $3.22X$ in respect with TensorFlow.**

***Index Terms*—distributed deep learning, benchmarking, Google TensorFlow, PyTorch, Apache MXNet**

I. Introduction

In the recent era, deep learning has become an asset in various fields, as artificial intelligence is constantly evolving [1]. For instance, neural networks are widely used in image classification [2] and recognition tasks [3], which also are of high interest in the medical domain [4]. Other areas of interest in deep learning include speech recognition [5], text classification [6] and emotion recognition [7]. Since deep learning finds applicability in all these totally different domains, many researchers have worked on improving this field.

The constant growth of available data has led researchers to design and exploit more complex network architectures to cover the data structure [8]. However, the vast amount of data available cannot be efficiently stored and used to train models in one single machine. Therefore, various distributed learning techniques have been proposed. The two most common approaches used to train deep neural networks are the all-reduce [9]–[11] and the parameter server one [12]–[14]. All-reduce training exploits reduce operators to combine the gradients from the various worker tasks that participate in the training process. Parameter server training uses server tasks that store a global model, which can be updated either in a synchronous or an asynchronous manner by various worker tasks.

All these different architectural approaches and the importance of the domain have opened the way to various tech companies to design and develop multiple systems for neural network training. Google has been developing TensorFlow [15] since 2015, while Facebook's AI Research lab has been working on PyTorch [16] since 2016. Other popular distributed learning systems include Apache MXNet [17], Theano [18], DeepLearning4J [19] and Chainer [20]. Moreover, specialized deep learning libraries for general-purpose systems have also been designed, as BigDL [21] for Apache Spark [22].

In this paper, we aim to provide an extensive performance evaluation on CPU clusters over the three most common distributed deep learning systems, namely Google's TensorFlow, Facebook's PyTorch and Apache MXNet. The choice of these particular three systems lies on covering all the aforementioned distributed architectures: all-reduce, synchronous parameter server and asynchronous parameter server. Our main contributions are the following:

- We quantify the performance gap of these systems using a variety of networks.
- We perform an extensive analysis of the results and present differences between the systems.

All the differences identified in this paper will help practitioners choose the right system depending on the training workload they want to run. The key findings of our analysis are:

- TensorFlow presents the slowest backward pass among the three systems, due to the implementations of some related operators. However, its forward pass is faster in most of the experiments and presents better reading mechanism, utilizing caches.
- MXNet is faster regarding the communication cost compared to the other systems, due to its custom RPC protocol implementation.
- PyTorch performs better on simpler network architectures, while MXNet should be preferred when training more complex networks.

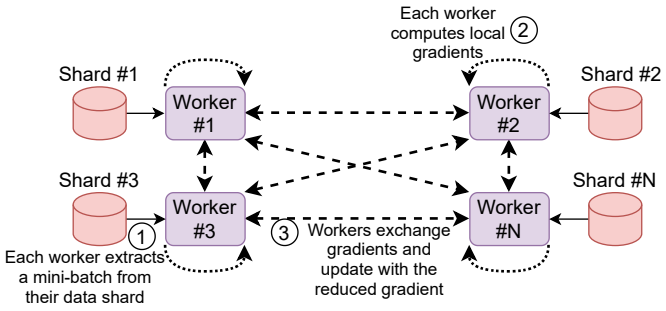


Fig. 1: All-Reduce Training

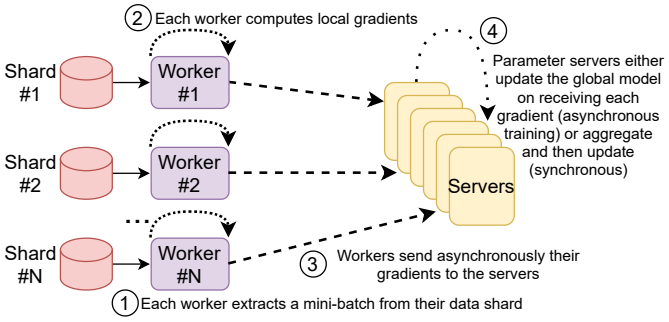


Fig. 2: Parameter Server Training

The rest of this paper is organised as follows: Section II provides an extesive background for the reader to understand the paper. Section III presents the architecture of the three systems examined under this experimental evaluation. Section IV and Section V describes the methodology and the results of our series of experiments. Finally, the paper concludes with a presentation of related papers to this work (Section VI) and a summary of our key insights and future work (Section VII).

II. Background

In this section, we present any backround knowledge for the reader to fully understand this paper. We provide some brief description regarding neural network training and an outline on the distributed training architectures of all-reduce and parameter server.

A. Neural Networks and Training

Neural networks are an algorithmic model which emulates the structure of a human brain, with sets of neurons (nodes) organized in hierarchical layers [1]. Their mathematical representation is a composite function of parameters, namely the network weights, which represent how the neurons affect their input before it is passed to the next layer. The training process of a neural network can be defined as an optimization problem, where we search for the optimal weight values that are able to minimize a *loss function*. The most common approach to solve optimization problems is to exploit the Gradient Descent algorithm. In the case of neural networks, the variant of the mini-batch Stochastic Gradient Descent [23] is preferred, due to the best tradeoff between the processing speed and the iterations before convergence. Backpropagation, the process of training a neural network, is summarized in the following steps :

1) A random mini-batch from the data is used to perform a *forward pass* on the network to obtain its output, *e.g.*, probabilities for classification tasks, which is the network's prediction for this mini-batch
2) The set of predictions is used to quantify the divergence from the real values, given the loss function, *e.g.*, categorical crossentropy for classification tasks.
3) A backward pass is used to obtain the gradients via the chain rule, which will be used to update the model parameters in the direction of the steepest gradient.

B. All-Reduce Training

One approach to achieve a data parallel distributed training relies on using All-Reduce techniques [24] on a peer-to-peer network of nodes [9]–[11]. Each node is considered a *worker*, who uses a local model copy to compute gradients from an assigned data shard. Note that each worker's shard differs from the one assigned to the others. The computed gradients are aggregated using all-reduce techniques before being used for updating each local model.

An iteration of All-Reduce training is outlined in Figure 1. Each worker extracts one mini-batch from its assigned data shard (Step 1), which is then used to compute their local gradients (Step 2). Having computed the local gradients, the workers exchange these values over network and, using All-Reduce techniques, they all finally have the aggregated gradients (Step 3). These aggregated gradients are used from each worker to update their local model. All-Reduce implies synchronized aggregations and, therefore, each worker has the same local model in the beginning of each training step with the others.

C. Parameter Server Training

Another well-known data parallel approach to train neural network in a distributed manner follows the parameter server approach [12]–[14]. In this setup, two types of tasks participate in the training: the *parameter servers* and the *workers*. Parameter servers are responsible for maintaining a global model, which will be updated with gradients computed from the workers either in a synchronous or an asynchronous mode. Workers exploit a local copy of the model to compute the gradients with mini-batches extracted from their local data shard.

Figure 2 illustrates a training step under the parameter server architecture. Having extracted the mini-batch from their local shard (Step 1), each worker computes a set of gradients (Step 2). Afterwards, these gradients are forwarded to the servers over the network (Step 3). Step 4 depends on the training mode. In asynchronous training, the servers update the global model directly after receiving a set of gradients from a worker and finally forward the new version of the model back to this worker. In case synchronization is preferred, servers wait to receive the updates from all workers, aggregate them and use the aggregated value to alter the global model. The model is finally forwarded to all workers.

TABLE I: System Specifications

System	Distributed Architecture		Data Loading	Lin. Alg. Library	Communication Library	High-Level Interface
	Synchronous Mode	Asynchronous Mode				
TensorFlow	All-Reduce	Parameter Server	Pipeline, Caching	Eigen	gRPC	Keras
PyTorch	All-Reduce	-	Pipeline, No caching	oneDNN	Gloo	Built-In
MXNet	Parameter Server	Parameter Server	Pipeline, No caching	oneDNN	custom RPC	Gluon

III. Systems

In this sections, we briefly discuss the main apsects of the three systems we benchmark in this paper, which are summarized in Table I. We only comment on characteristics that affect the performance of the systems.

A. Google TensorFlow

TensorFlow [15] is open-source machine learning platform which has been developed by Google since 2015.

Programming Model. TensorFlow's original model used dataflow graphs to model the training process. Tensors are used to represent the data. In the graph, nodes represent basic operations, *e.g.*, a sum or a matrix-vector multiplication and edges are used to transfer tensors between operations. Graphs are optimized and run under sessions. Latest TensorFlow versions enable the dynamic *eager execution*, which actually follows the imperative programming model, where each operation is computed inline. Graph execution is utilized on user demand to achieve optimizations on specific parts of the code.

Distributed Execution. TensorFlow supports both types of distributed learning described in sections II-B and II-C. Synchronous training is performed using an all-reduce variant called Ring-Reduce [10]. Asynchronous training can be achieved under the parameter server architecture.

Data Access. TensorFlow offers *Dataset API* in order to fetch data from disk which is used to create a pipeline of operations essential to read, decode, shuffle, augment, cache and extract the data in the form of mini-batches. Data related operations are performed in parallel with the process of the previous mini-batch. Depending on the training mode, TensorFlow reads different amount of data. Synchronous TensorFlow requires each worker to read only the shard assigned to them. In the case of asynchronous TensorFlow, according to the official tutorial [25], each worker reads the whole dataset, as opposed to the synchronous case, but processes only the one third in each epoch.

Linear Algebra Library. TensorFlow facilitates the Eigen [26] library. Eigen provides header for a set of linear algebra operators, *e.g.*, matrix and vector related operators.

Communication Protocol. TensorFlow exploits gRPC, a general-purpose RPC infrastructure initially created by Google, for the workers to communicate irrespective of the training mode.

High-Level Interface. TensorFlow encapsulates the Keras [27] library for building deep learning models easily.

B. PyTorch

In 2016, Facebook's AI Research Lab initially released PyTorch [16], an open-source machine learning framework.

Programming Model. PyTorch utilizes, as TensorFlow, a dynamic imperative programming model by providing an array-based programming model. Data are processed in the form of PyTorch tensors and PyTorch programs are executed in an eager manner.

Distributed Execution. PyTorch distributed execution relies on the all-reduce training type, supporting synchronous training. PyTorch offers multiple modules related with distributed execution, with Distributed Data-Parallel Training being the one used for our experiments. Regarding all-reduce, PyTorch groups the gradients in buckets from the network's output to the top and when all the gradients in a bucket have been computed, the all-reduce process is initiated for this bucket in the form of Ring-Reduce. However, both the forward and the backward pass serve as synchronization points. It should be noted that even though asynchronous training is possible with PyTorch, it is not fully supported yet and thus is omitted from our experimental evaluation.

Data Access. Regarding data reading, PyTorch offers *DataLoader* which reads data in the form of mini-batches in parallel with the training. Data could be augmented before mini-batch extraction. No caching mechanism is available in the *DataLoader* mechanism.

Linear Algebra Library. Linear algebra operations are performed using oneDNN [28] library, which is optimized for Intel Processors.

Communication Protocol. Multiple protocols and libraries are supported for the processes to communicate with each other. However, in the case of CPU clusters, Gloo [29] communication library is the default option, which support point-to-point and collective operations

High-Level Interface. PyTorch provides its own built-in deep learning building blocks to design neural networks.

C. Apache MXNet

Apache MXNet [17] was released in 2016 and joined the Apache Software in 2017. It is a flexible and efficient library for deep learning.

Programming Model. MXNet provides both a declarative and an imperative style of programming. In declarative programming, a computational graph is defined using *Symbols*, which can maintain an internal *state*. Before executing the computational graph, MXNet optimizes the graph in terms of performance and assigns memory to the variables. For imperative programming, *NDArray* is offered as a structure.

Distributed Execution. Apache MXNet adopts the parameter server architecture to provide both synchronous and asynchronous training. It exploits a key-value store structure, named the *KVStore*, which supports push and pull operations.

TABLE II: Neural Network Characteristics

Network	#Layers	#Parameters
LeNet-5	5	60K
AlexNet	8	62M
ResNet-18	18	11M
ResNet-50	50	25M

Parameter servers are responsible to handle synchronization issues. KVStore is also responsible to handle gradient aggregation. Except for the servers and workers, a task named *scheduler* is used to set up the cluster.

Data Access. MXNet provides built-in data loading modules, which are able to operate in parallel with training, offering various capabilities, *e.g.*, mini-batch formation, augmentation *etc.*

Linear Algebra Library. MXNet uses the same library as PyTorch for linear algebra operations, the oneDNN.

Communication Protocol. Regarding process communication, the system utilizes a custom implementation of the RPC protocol.

High-Level Interface. For easy model design, MXNet exploits Gluon [30], a simple and accurate API for build deep learning models.

IV. Experimental Setup

In this section, we discuss our experimental setup and the methodology used to gather our results.

A. Hardware and Software Configuration

Our experimental evaluation is performed on a cluster consisting of three virtual machines. Each virtual machine has 4 virtual CPUs (Intel Core Processor @ 2.2 GHz), 16 GB of RAM and 30 GB HDD. Since all benchmark are Python programs, all VMs have Python 3.6.9 installed. Regarding the versions of the systems we benchmark, we use TensorFlow 2.5.0, PyTorch 1.7.0 and MXNet 1.7.0.

All systems are tested with all their available training modes, *i.e.*, synchronous and asynchronous (except PyTorch - see section III-B). For All-Reduce training setups we deploy one worker task on each machine. When training under the parameter server architecture, both a parameter server and a worker task are executed on each virtual machine.

B. Benchmarks

In our experimental evaluation, we use four well-known convolutional neural networks from the image classification domain. Specifically our implementation include two networks with simple structure, LeNet-5 [31] and AlexNet [2], and two with more complex units, which belong to the ResNet architecture [32], *i.e.*, ResNet-18 and ResNet-50. Table II provides information regarding the size of these networks. We have utilized the high-level interface offered by each system to implement these networks.

The datasets upon which the neural networks were trained are CIFAR-10 and CIFAR-100 [33]. Both of them consist of 60,000 labeled images of 32x32 dimensions and 3 channels. The difference between the two is located at the number of classes, as they contain 10 and 100 classes respectively. Each worker has the shard of the data they need in their local storage.

TABLE III: Model-dataset combinations used.

Neural network	Dataset
Lenet-5	CIFAR-10
AlexNet	CIFAR-10
ResNet-18	CIFAR-10
ResNet-18	CIFAR-100
ResNet-50	CIFAR-10
ResNet-50	CIFAR-100

TABLE IV: Training parameters.

Parameter	Value
Dataset Size	50,000
Global batch size	126
Local batch size	42
Steps per epoch	396
Number of epochs	10
Input size	32x32 (64x64 only for AlexNet)

Benchmarks were constructed by training the two datasets with the networks. In total, we performed six different training processes under all the systems and training modes available. The benchmarks, as model-dataset combinations are outlined in Table III. Since we want only to quantify the performance gap of the systems, we run the benchmarks for a fixed number of epochs (10). In general, neural networks are usually run for a fixed number of epochs, unless some specific convergence technique, *e.g.*, early stopping is defined. Therefore, regarding the performance gap, it is safe to run a few number of epochs to identify architectural and implementation issues. Other training related parameters are described in Table IV. Note that images were used at their original size (32x32) as input in the networks, with the exception of AlexNet where, due to the increased amount of sampling taking place, the dimensions used where 64x64. Thus, images were resized at first for this experiment.

C. Methodology

Our experimental evaluation is conducted in three phases. The main phase is the distributed training phase. During that phase the benchmarks (see section IV-B) were run under all the systems and training modes in order to record their corresponding execution time.

The second phase of this benchmarking included the breakdown of the execution time in stage times, *e.g.*, time needed for the forward and the backward pass. Using appropriate profilers for each system, *e.g.*, TensorBoard for TensorFlow, we identified the time spent by each system on the various stages and operations. Communication cost is identified as the difference between the total execution time and the sum of the forward pass, backward pass and data loading time.

The final phase of our work includes the process of further analyzing profiling data in order to extract any performance related issues, *i.e.*, heavy operations. In this way, we can identify any reason behind the overhead of a specific stage.

V. Experimental Evaluation

In this section, we provide our detailed experimental evaluation to identify the performance gap between the three systems.

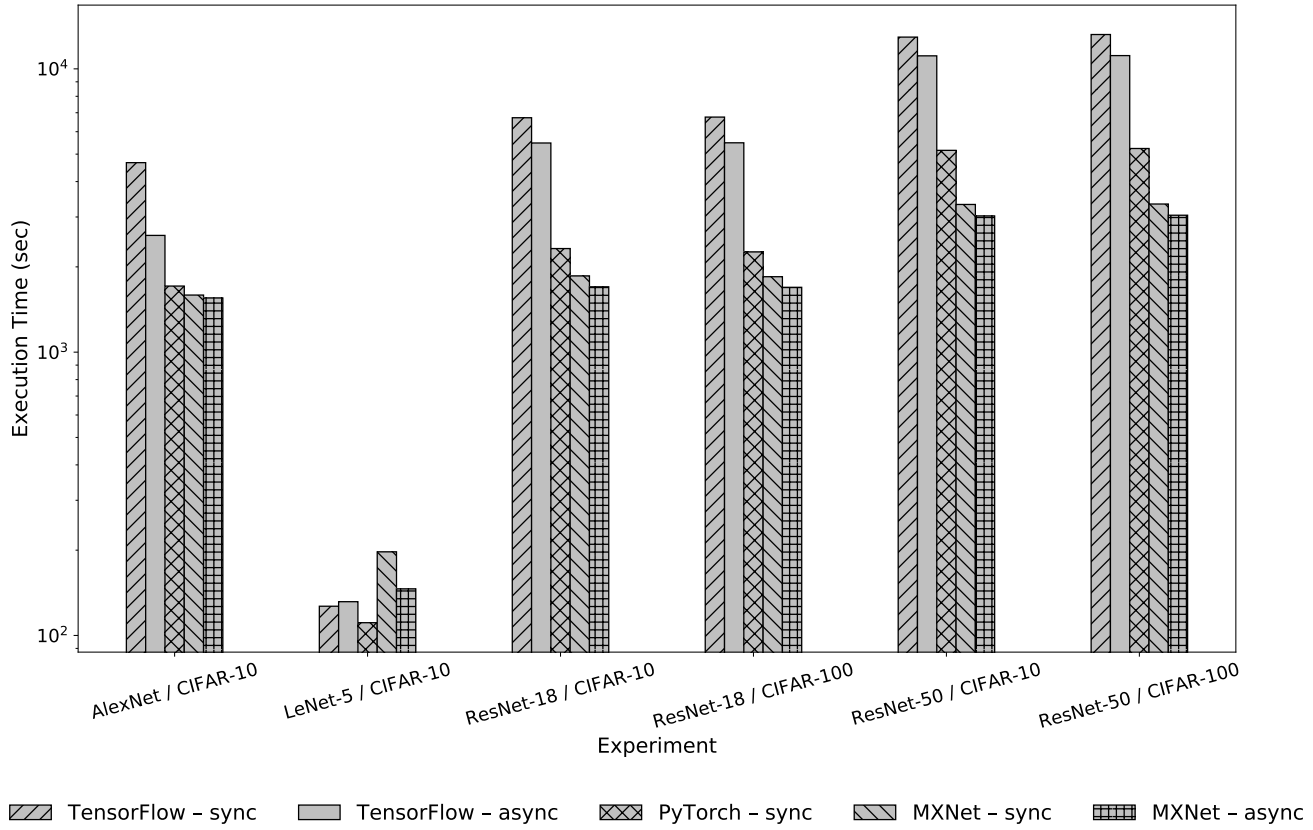


Fig. 3: Execution times of TensorFlow, PyTorch and MXNet on benchmarks. TensorFlow and MXNet present execution times with both synchronous and asynchronous learning. Y-Axis is presented in logarithmic scale.

We present our analysis as follows: Section V-A presents a first comparison over the total execution time, where we identify the overall performance gap between the systems. In sections V-B, V-C and V-D, we discuss in detail, exploiting the profiling data, how the performance gap between the systems occurs. Note that in each case the execution time is initially broken down to the loading time , the forward pass time, the backward pass time and the communication time. As loading time, we identify disk I/O, data decoding and transformations that do not take place in parallel with a forward or backward pass phase. Communication time refers to data exchange and any synchronization overheads that may occur. Finally, in sections V-F and V-G we comment on the factors affecting the communication and data loading cost for the systems respectively.

Our main findings are summarized as follows:

- In Synchronous Parameter Server experiments less time is spent over network communication compared to equivalent all-reduce experiments.
- TensorFlow is superior in data loading thanks to caching mechanisms, while MXNet performs better at network communication up to $3.19X$ due to each custom RPC implementaion.
- TensorFlow's forward pass is faster in most of the experiments. However its backward pass is crucially affected by the operator *convolution_backward*, which stalls up to $7.67X$ more compared to the counterpart operators of the other systems. The performance of PyTorch is crucially hurt by the operators *batch_norm* and *batch_norm_backward*, which are $29.7X$ and $21.4X$ slower than the equivalent MXNet operators.

A. Overall Comparison

Figure 3 illustrates the execution time of each system and training mode for the series of experiments conducted. Overall, we notice that TensorFlow seems to be the slowest of all, while MXNet presents the best performance. The only experiment that does not confirm this observation is when training the LeNet-5 network, where PyTorch is the fastest and MXNet lags the other two systems irrespective the training mode.

Regarding the synchronous case, TensorFlow, which presents the poorest performance, is by average $2.65X$ and $3.66X$ slower compared to PyTorch and MXNet respectively. TensorFlow largest performance gap from the other two systems is identified on ResNet experiments, where it lags PyTorch and MXNet by $2.99X$ (ResNet18 / CIFAR100) and $3.97X$ (ResNet-50 / CIFAR-100) respectively. This behaviour is attributed to specific operators, as we will discuss in section V-D.

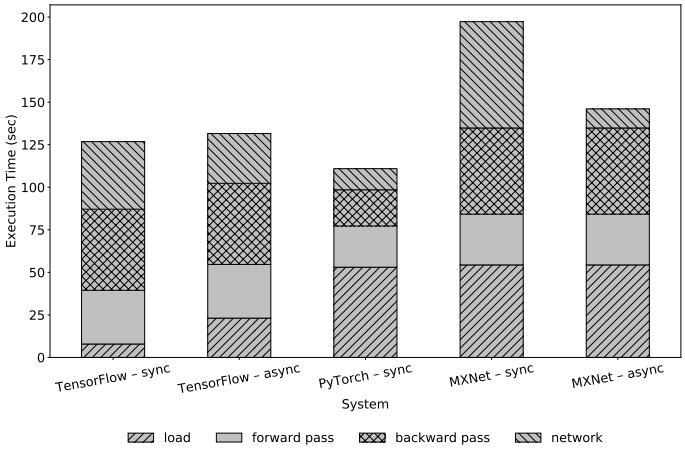


Fig. 4: Breakdown of Execution Time (LeNet-5 / CIFAR-10)

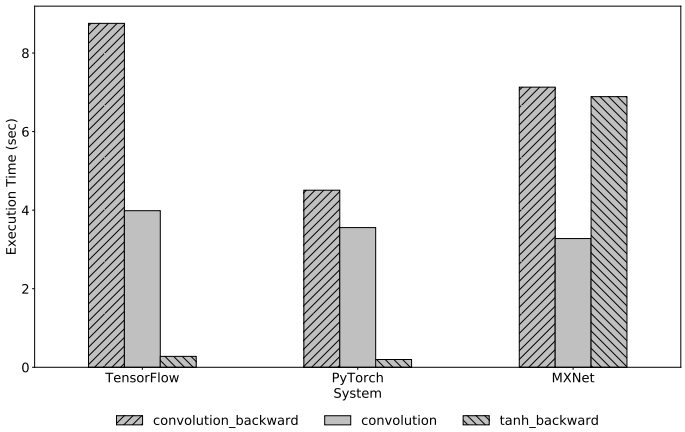


Fig. 5: System Profiling Data (LeNet-5 / CIFAR-10)

When training under an asynchronous parameter server environment, MXNet clearly outperforms TensorFlow by average $3.22X$, where the gap meets its greatest value on ResNet experiments, as in the synchronous case. In LeNet-5, the two systems present similar execution times.

B. LeNet-5

Figure 4 presents a breakdown of the execution times for this experiment. Note that the only case where TensorFlow is not the slowest of all, as discussed in V-A, is that of LeNet-5. This is mainly due to the data loading part, which will be further discussed in the data loading section (V-G). Overall, it is worth noticing that TensorFlow is $1.56X$ faster than MXNet in synchronous mode.

In the previous paragraph, we mentioned that the only neural network in which TensorFlow does not rank last is LeNet-5. It is also worth noting that this is only series of experiments (section V-A) where MXNet does not occupy the first position, but the last one. It is interesting to identify why PyTorch outperforms MXNet by $1.8X$ in this case. A close look in Figure 4 indicates that this can be mainly attributed to the backward pass time. To further understand the related issues, we present profiling data from this experiment in Figure 5. The backward pass operator that makes the difference is that

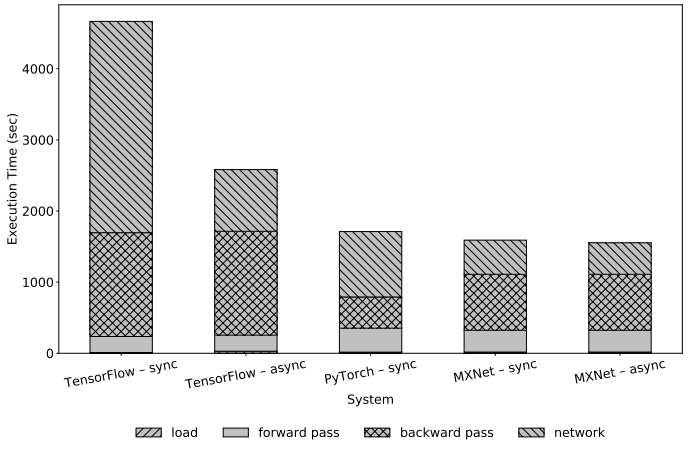


Fig. 6: Breakdown of Execution Time (AlexNet / CIFAR-10)

of *tanh_backward*, *i.e.*, is the backward pass on the activation function tanh. For this operator, MXNet is actually $\times 34.8$ and $\times 24.8$ slower than PyTorch and TensorFlow in this experiment. It is worth noting that TensorFlow's backward pass performance is hurt by the *convolution_backward* operator and, therefore, cannot surpass PyTorch in total, even if it presents better loading time.

Another interesting observation is that synchronous TensorFlow is faster than asynchronous in this experiments. This may initially sound like a paradox, since synchronization adds an extra overhead to the communication time. However, this can be supported if we consider the way the amount of data read by each worker nodes in each experiment in combination with the small size of this neural network. In the synchronous mode, each worker loads into memory the part of the dataset that assigned to them (one third in our case) and processes the same subset of data throughout the training. In contrast, in asynchronous mode, each worker loads the entire dataset, and processes the batch routed by the coordinator at a time, as discussed in section III-A. Therefore, while in the asynchronous mode there is a little better shuffling of the data, since any batch can occur at any node, there is also a higher cost of loading the data since the whole datasets is read from each worker. The reading phase asynchronous TensorFlow induces $2.9X$ overhead compared to the synchronous one, which cannot be eliminated by the less communication cost, where the time spend in only by $1.35X$ less.

C. AlexNet

Figure 6 outlines the breakdown of the execution time for the conducted experiments with AlexNet benchmark. Overall, TensorFlow appears to be the slowest in the series of experiments, while PyTorch and MXNet present a similar performance. When using synchronization, TensorFlow's speed is $2.73X$ and $2.93X$ less from PyTorch and MXNet respectively. In the asynchronous setup, MXNet is superior to TensorFlow by $1.66X$.

TensorFlow's performance appears to be crucially affected mainly by the backward pass and the communication time. For instance, the backward pass of PyTorch is $3.32X$ faster

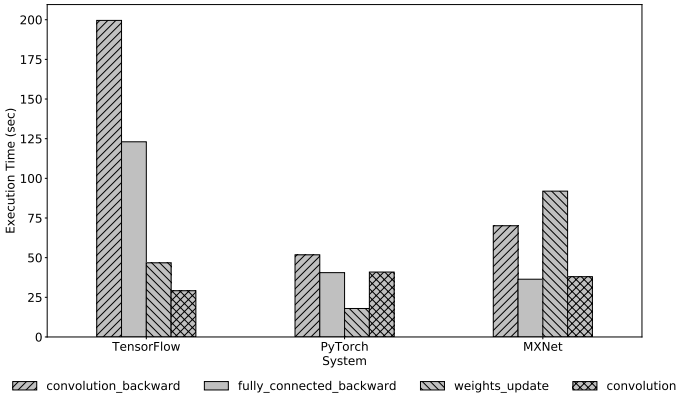


Fig. 7: System Profiling Data (AlexNet / CIFAR-10)

than the equivalent TensorFlow phase. Regarding the time spend over the network, TensorFlow is especially hurt in the synchronous case, where it suffers also the synchronization overheads. Specifically, compared to synchronous MXNet, this phase is $6.18X$ slower.

It is observed that, although PyTorch and MXNet are close in terms of total time (MXNet only 1.1X faster), the former excels in the backward pass while the second in communication time. In general, in the case of AlexNet we identify that MXNet is far superior to the other two in terms of communication time, irrespective of the mode. We will provide a general discussion over the communication time in section V-F.

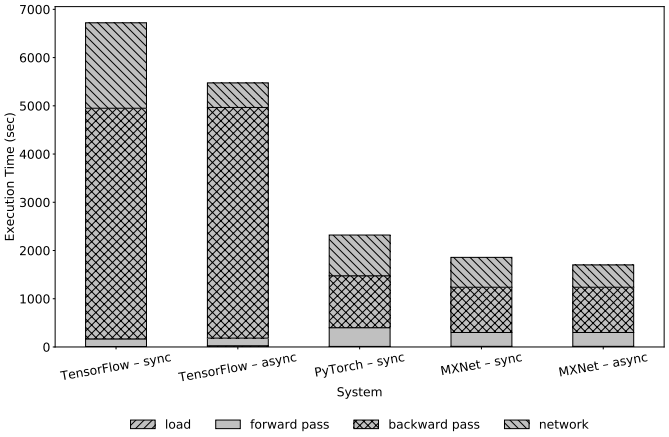
Since both MXNet and PyTorch are based on the oneDNN library, it is interesting to identfy where this performance gap lies on. This operator stalling MXNet, according to figure 7 is *weights_update*. This operator is responsible for updating the weights of the neural network with the calculated gradients. In particular, the corresponding PyTorch operator is $5.1X$ faster. Another operator which favors PyTorch is the *convolution_backward* one, which lags PyTorch over MXNet by $1.35X$. The rest operators presented in Figure 7 present similar performance between these two systems.

Another interesting observation lies on how TensorFlow's backward pass is affected. Apart from the *weights_update* operator, where MXNet presents the worst performance, all the backward pass operators meet their worst performance in TensorFlow. Specifically, both the *convolution_backward* and the *fully_connected_backward* are $3.85X$ and $3.03X$ slower from the equivalent PyTorch operators respectively.

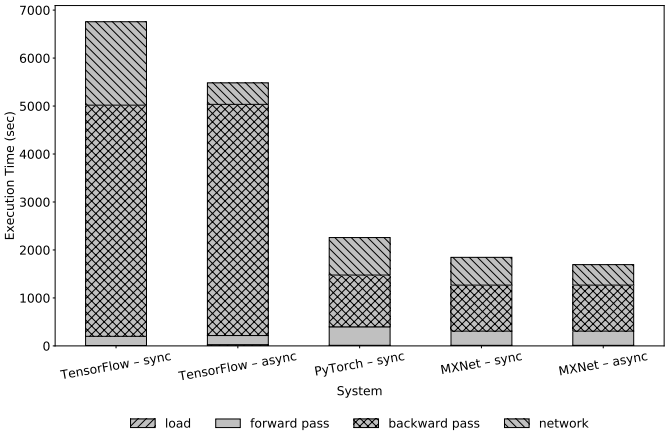
While TensorFlow suffers from its backward phase, we discussed earlier that its forward phase presents better performance compared to the other two systems. For instance, in Figure 7, we ascertain that the *convolution* operator is by average $1.35X$ faster in TensorFlow in respect to the other two systems.

D. ResNet-18 and ResNet-50

Figures 8a and 8b present the breakdown of the execution times for training ResNet-18 on CIFAR-10 and CIFAR-100 respectively. Following the same pattern as the AlexNet experiments presented in section V-C, MXNet, in every training



(a) CIFAR-10



(b) CIFAR-100

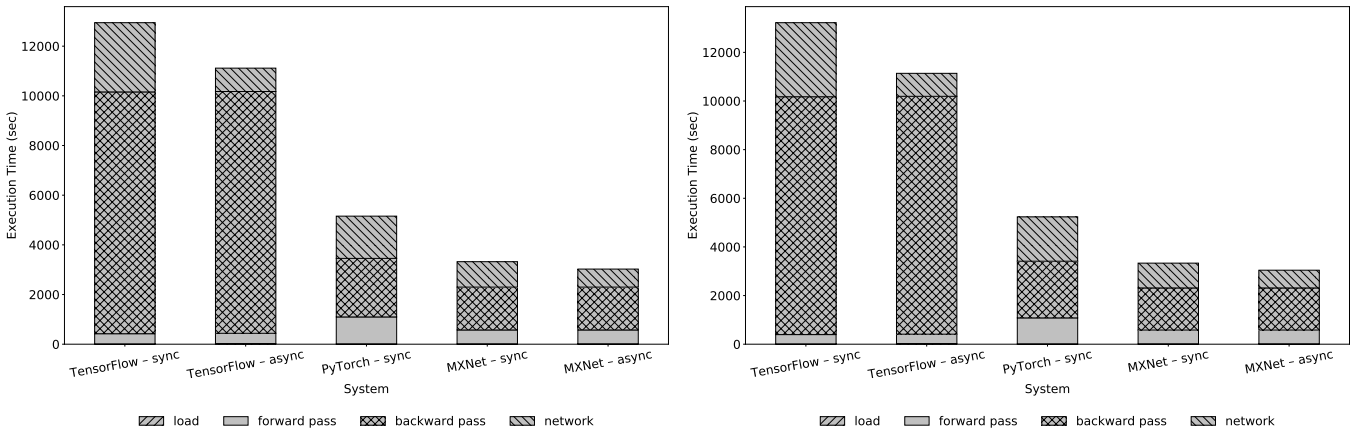
Fig. 8: Breakdown of Execution Time for ResNet-18 networks on CIFAR-10 (a) and CIFAR-100 (b) datasets

mode, is again faster than PyTorch up to $1.37X$. TensorFlow also presents the worst performance with the gap being at $3.65X$. This gap is identified between synchronous TensorFlow and synchronous MXNet.

A closer look on Figure 8 affirms the assumption that TensorFlow suffers from its backward phase. Synchronous TensoFlow is also crucially affected by the time spent over the network.

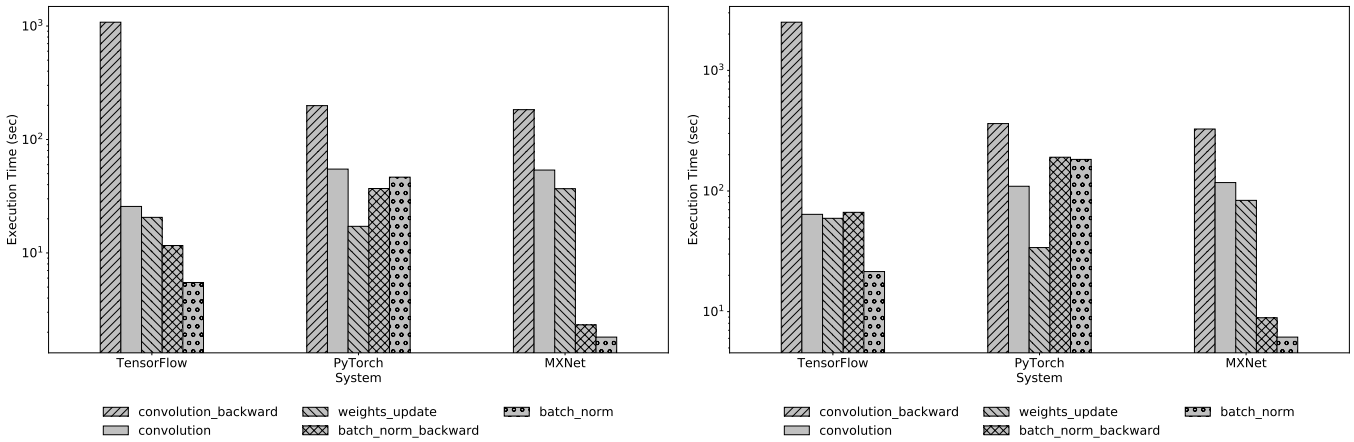
PyTorch appears to perform worse compared to MXNet. Its performance seems to be affected more by time spent in network and the forwards pass. The forward pass induces $1.34X$ in respect to MXNet (average of training both with CIFAR-10 and CIFAR-100). Comparing with synchronous MXNet, which also faces related overheads, PyTorch spends $1.36X$ more time over the network. Regarding the backward pass, MXNet and PyTorch present similar performance with a smaller than $1.2X$.

All the above claims and observations are further confirmed with the series of experiments that refer to ResNet-50. The corresponding results are illustrated in Figure 9 for both the datasets. TensorFlow's worst performance is located in this experiment with the performance gap reaching the value of $3.96X$ compared to MXNet's best performance on the CIFAR-



(a) CIFAR-10 (b) CIFAR-100

Fig. 9: Breakdown of Execution Time for ResNet-50 networks on CIFAR-10 (a) and CIFAR-100 (b) datasets



(a) ResNet-18 (b) ResNet-50

Fig. 10: System Profiling Data (ResNet / CIFAR-10)

100 dataset (Figure 9b). We also notice larger performance gap ($1.6X$) between PyTorch and MXNet, occurring from the forward pass ($1.9X$), the backward pass ($1.4X$) and the communication time ($1.78X$ from synchronous MXNet).

In order to identify the reasons behind the delay of TensorFlow and PyTorch, in Figure 10 we present the execution time of some interesting operators from the profiling process of training both ResNet-18 and ResNet-50 on CIFAR-10. The main operator responsible for the performance gap of PyTorch and MXNet in the forward pass is the *batch_norm*, *i.e.*, . the normalization layer, which is up to $29.7X$ slower for PyTorch. The same pattern is identified also at the backward pass. In this case, the corresponding operator, *i.e.*, *batch_norm_backward* is responsible for this difference, since the PyTorch implementation is by average $21.4X$ slower compared to MXNet. It can therefore be observed that MXNet provides a much faster implementation for this layer in both forward and backward pass. TensorFlow's backward pass is affected again by the *convolution_backward* operator, while the equivalent operator for the forward pass, *i.e.*, *convolution*, performs better compared to the implementations of the other two systems.

E. General Discussion on Operators

The slow backward pass observed in TensorFlow is, as mentioned in the previous sections, the main reason why it lags behind the other two in almost all experiments. TensorFlow uses the Eigen library to implement its operators. On the other hand, PyTorch and MXNet are based on oneDNN (formerly known as MKL-DNN). The oneDNN provides implementations of deep learning operators that are specifically optimized to run on Intel processors. Since all the experiments in the current work were conducted on a cluster consisting of Intel CPUs, certainly the systems based on oneDNN had a strong lead. This is moreover evident from the profiling of the operators discussed on the analysis of the experiments. In all cases, the *convolution_backward* operator, which is usually the most time-consuming operator in convolutional neural networks, appears to be much slower in the TensorFlow implementation. This particular TensorFlow operator ends up

TABLE V: Mean Communication Time for each System

Synchronous			Asynchronous	
TensorFlow	**PyTorch**	**MXNet**	**TensorFlow**	**MXNet**
1741.81	869.51	546.19	586.65	410.54

running for $7.67X$ the time the MXNet counterpart takes in the ResNet experiments.

It is of course worth noting that for all neural networks except LeNet-5, TensorFlow has a faster operator in terms of forward pass convolution, as discussed in sections V-C and V-D. However, despite the faster implementation of Eigen for one of the most important forward pass operators, it is the backward pass operators that have the largest impact, and thus PyTorch and MXNet are faster in the majority of experiments.

F. Communication Cost and Network

As we defined back in the introduction of this section, in the asynchronous training experiments, communication time refers to the time consumed in data transfer over the network. For the synchronous case, this time also includes any additional time spent waiting for the slower worker to finish processing.

According to almost all experiments (except LeNet-5 where the numbers are too small to draw a clear conclusion) the fastest communication seems to be MXNet. Particularly in the case of AlexNet (Section V-C), while in the other two systems the communication cost seems to scale with increasing parameters (see table II), MXNet does not seem to be crucially affected and even has a strong lead in this experiment thanks to the reduction in communication cost. In particular, in AlexNet, although MXNet has slower backward pass, it has $\times 1.9X$ faster communication than PyTorch.

Table V presents the average of the time spent over the network from all the experiments run on the CIFAR-10 dataset for each system. MXNet is affirmed to spent in average the least time compared to the other two. While in the asynchronous Parameter Server architecture, there is not a large performance gap between asynchronous MXNet and TensorFlow, MXNet need $1.42X$ less time. MXNet's behaviour over the network can be attributed to the custom implementation of the RPC protocol it provides via a launch module versus the gRPC used by TensorFlow. Regarding the synchronous case, PyTorch (Gloo) appears to have better all-reduce related operators compared to TensorFlow (gRPC), since it is $2X$ faster. However, MXNet's synchronous parameter server seems to operate better than the all-reduce mechanisms exploited by TensorFlow and PyTorch.

G. Data loading and Memory Consumption

In terms of loading the dataset into memory for processing each batch, there are two approaches followed by the three systems in question. In the first, each batch of data is loaded and then cached so that, the next time it is needed (the next epoch usually) it will be loaded into memory much faster. In the second approach, every batch that is needed is loaded directly from disk, no matter how many times the same batch is requested. In this approach there is lower consumption of node resources, but slower data loading.

TensorFlow uses caching, while PyTorch and MXNet take the second approach, and as a result TensorFlow requires more memory. TensorFlow uses more memory in asynchronous mode than in synchronous mode, as mentioned in section V-B. Since all three systems load batches of data in parallel with their processing, loading batches from disk does not add much latency for large neural networks (where we also have large processing time, i.e., forward and backward pass). In the case of LeNet-5, however, loading each batch is of the same order of magnitude as processing the batch. Hence it does not make good use of the parallel loading of consecutive batches and therefore adds a delay for systems that do not use the cache. This delay, given that fact that for small networks data loading is an important part (in terms of time) of network training, gives TensorFlow a strong advantage over the other two.

VI. Related Work

Over the last years, multiple works have evaluated deep learning systems. In 2016, Fathom [34] was released as a set of workloads for benchmarking. However, this suite was designed especially to evaluate TensorFlow. In 2017, the authors provide in [35] an extensive experimental evaluation even for multinode setups. However, they examine different systems instead of PyTorch and MXNet. They also do not perform a profiling analysis over the performance gap to identify any heavy operations. Dawnbench [36] was also released in 2017, as a benchmark aiming to measure the execution time of training a network to the state-of-the-art accuracy. They also study how various network related optimizations affect the performance. In 2018, an experimental analysis [37] on how deep learning frameworks operate in distributed setups was released, but it measured the performance on GPUs and GPU clusters and did not cover CPU clusters. Another study [38] regarding GPU clusters was performed in 2018, where the authors also offered a detailed mathematical model to describe the performance. In 2019, we conducted a detailed experimental evaluation [39] to quantify the performance gap of general-purpose and specialized systems, where we also studied some machine learning workloads. However, we did not examine deep learning cases, and only TensorFlow was evaluated with Spark. Last year, a theoretical survey [40] without an experimental evaluation was published, discussing various aspects of multiple systems, including the three we examine in this paper.

VII. Conclusions and Future Work

In this paper, we provided a detailed experimental evaluation to quantify the performance gap between three widely used deep learning systems: Google TensorFlow, Facebook PyTorch and Apache MXNet. Our series of experiments indicated that TensorFlow lags behind both PyTorch and MXNet either in synchronous or asynchronous learning setups. TensorFlow's main stalling parameter is its backward pass phase, attributed to the operator *convolution_backward*. PyTorch is kept behind MXNet due to operators related with batch normalization. Overall, MXNet presents the best performance and the best

communication times due to its custom RPC implementation. In general, for synchronous setups, Parameter Server is found to perform better than the all-reduce alternative.

The experimental analysis presented in this paper can be extended to multiple directions. We aim to extend our experimental evaluation to other workloads beyond image classification and systems. We also aim to study other aspects of the presented systems and extend our research to quantify and explain the performance gap on GPU clusters.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] T. Yu, J. Meng, and J. Yuan, "Multi-view harmonized bilinear network for 3d object recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 186–194.

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[5] D. Amodei, S. Ananthanarayanan, R. Anubhai *et al.*, "Deep speech 2 : End-to-end speech recognition in english and mandarin," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 173–182. [Online]. Available: https://proceedings.mlr.press/v48/amodei16.html

[6] W.-C. Chang, H.-F. Yu, K. Zhong *et al.*, "Taming pretrained transformers for extreme multi-label text classification," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3163–3171.

[7] A. S. Khan, Z. Li, J. Cai *et al.*, "Group-level emotion recognition using deep models with a four-stream hybrid network," in *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, 2018, pp. 623–629.

[8] N. Provatas, I. Konstantinou, and N. Koziris, "Towards faster distributed deep learning using data hashing techniques," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 6189–6191.

[9] M. Cho, U. Finkler, and D. Kung, "Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning," in *Proceedings of the 2nd SysML Conference*, 2019.

[10] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[11] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[12] J. Dean, G. Corrado, R. Monga *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, p. 1223–1231.

[13] M. Li, L. Zhou, Z. Yang *et al.*, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, vol. 6, 2013, p. 2.

[14] M. Li, D. G. Andersen, J. W. Park *et al.*, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.

[15] M. Abadi, P. Barham, J. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, p. 265–283.

[16] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[17] T. Chen, M. Li, Y. Li *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[18] H. Ma, F. Mao, and G. W. Taylor, "Theano-mpi: a theano-based distributed training framework," in *European Conference on Parallel Processing*. Springer, 2016, pp. 800–813.

[19] D. Deeplearning4j, "Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0," 2016.

[20] S. Tokui, R. Okuta, T. Akiba *et al.*, "Chainer: A deep learning framework for accelerating the research cycle," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2002–2011.

[21] J. J. Dai, Y. Wang, X. Qiu *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[22] M. Zaharia, R. S. Xin, P. Wendell *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2934664

[23] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[24] M. Barnett, L. Shuler, R. van De Geijn *et al.*, "Interprocessor collective communication library (intercom)," in *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 1994, pp. 357–364.

[25] "Parameter server training with parameterserverstrategy." [Online]. Available: https://www.tensorflow.org/tutorials/distribute/parameter_server_training

[26] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[27] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[28] Oneapi-Src, "oneapi-src/onednn: oneapi deep neural network library (onednn)." [Online]. Available: https://github.com/oneapi-src/oneDNN

[29] Facebookincubator, "Gloo: Collective communications library with various primitives for multi-machine training." [Online]. Available: https://github.com/facebookincubator/gloo

[30] R. Dathathri, G. Gill, L. Hoang *et al.*, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 752–768. [Online]. Available: https://doi.org/10.1145/3192366.3192404

[31] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[33] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[34] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[35] S. Shams, R. Platania, K. Lee, and S.-J. Park, "Evaluation of deep learning frameworks over different hpc architectures," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1389–1396.

[36] C. A. Coleman, D. Narayanan, D. Kang *et al.*, "Dawnbench: An end-to-end deep learning benchmark and competition," in *NIPS ML Systems Workshop*, 2017.

[37] H. Zhu, M. Akrout, B. Zheng *et al.*, "Benchmarking and analyzing deep neural network training," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 88–100.

[38] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 949–957.

[39] E. Kassela, N. Provatas, I. Konstantinou *et al.*, "General-purpose vs. specialized data analytics systems: A game of ml & sql thrones," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 317–326.

[40] R. Mayer and H.-A. Jacobsen, "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–37, 2020.