Parallelized Training of Deep NN – Comparison of Current Concepts and Frameworks

Sebastian Jäger
inovex GmbH
76131 Karlsruhe, Germany
sebastian.jaeger@inovex.de

Hans-Peter Zorn
inovex GmbH
76131 Karlsruhe, Germany
hans-peter.zorn@inovex.de

Stefan Igel
inovex GmbH
76131 Karlsruhe, Germany
stefan.igel@inovex.de

Christian Zirpins
Karlsruhe University of Applied Sciences
76133 Karlsruhe, Germany
christian.zirpins@hs-karlsruhe.de

ABSTRACT

Horizontal scalability is a major facilitator of recent advances in deep learning. Common deep learning frameworks offer different approaches for scaling the training process. We operationalize the execution of distributed training using Kubernetes and helm templates. This way we lay ground for a systematic comparison of deep learning frameworks. For two of them, TensorFlow and MXNet we examine their properties with regard to throughput, scalability and practical ease of use.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Distributed algorithms**; • **Information systems** → *Computing platforms*;

KEYWORDS

Deep Learning Frameworks, Deep Neural Networks, Distributed Parallelized Training, Kubernetes

ACM Reference Format:
Sebastian Jäger, Hans-Peter Zorn, Stefan Igel, and Christian Zirpins. 2018. Parallelized Training of Deep NN – Comparison of Current Concepts and Frameworks. In *Second Workshop on Distributed Infrastructures for Deep Learning (DIDL '18), December 10–11, 2018, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3286490.3286561

1 INTRODUCTION

Deep learning (DL) methods provide solutions for a number of difficult problems like image or speech recognition and are therefore being utilized in a growing number of individual application systems. Optimizing respective deep neural networks (DNN) requires timely data analysis for individual models with low update latency while facing an increasing size of models and training data. A common way to achieve this in a scalable way is to distribute and parallelize the training process.

To ease implementation, specific frameworks have emerged as specialized middleware for DL applications. DL frameworks are available in many varieties differing, e.g., in supported DL methods, technology platform/stack and vendor/ecosystem. As explained above, the ability of DL frameworks to parallelize and distribute training of DNN is crucial and practitioners need to assess their ability to scale appropriately.

Accordingly, we study the performance of two DL frameworks, TensorFlow[1] and MXNet[2], with respect to parallelized and distributed training of convolutional (CNN) and recurrent (RNN) neural networks. A particular goal is to examine these frameworks in a common and easily accessible setting that makes our results applicable to a wide range of practitioners.

To this end, we utilize a state-of-the-art cloud infrastructure and technology stack based on Kubernetes[3] and the Helm[4] package manager. Herein, we evaluate the horizontal scalability of data parallelism mechanisms for multi-machine setups with CPU-based training.

As part of our contribution we survey the features of DL frameworks with respect to parallelized DNN training and discuss related performance studies. Furthermore, we present the results of our own performance and scalability benchmarks and discuss practical experiences that we have gained with the parallelization mechanisms of two popular DL frameworks. Thus, our work can support practitioners to take decisions in the design process of DL infrastructure based on widely accessible cloud platforms.

The rest of the paper is structured as follows. Section 2 gives an overview of background concepts for parallelizing the training of DNNs. Section 3 introduces principles of DL frameworks focusing on distributed training and illustrates them by means of TensorFlow and MXNet. Section 4 presents our experimental study on performance and scalability of distributed training with TensorFlow and MXNet on a Kubernetes cloud platform. Finally, section 5 discusses related work and section 6 concludes.

DIDL '18, December 10–11, 2018, Rennes, France

ACM ISBN 978-1-4503-6119-4/18/12…$15.00
https://doi.org/10.1145/3286490.3286561

[1] https://www.tensorflow.org
[2] https://mxnet.apache.org
[3] https://kubernetes.io
[4] https://helm.sh

2 CONCEPTS OF DISTRIBUTED TRAINING

Deep neural networks consist of thousands to million parameters and require a significant amount of data to learn these parameters. The training is a computationally intensive and time-consuming process. The most important way to accelerate the training is to parallelize and distribute computation across multiple devices and machines. In this section, we describe the principles and concepts of different distributed training methods.

2.1 Model Parallelism

The idea of model parallelism is to split the model across different processing units and use the same data for each part of the model. The main advantage is the possibility to train extremely large models which would not fit into the memory of one device. However, the efficiency of model parallelism depends heavily on the architecture and way the model is split. Figure 1 shows different examples. As on the left side of figure 1 is shown, there is no good way to use model parallelism for fully connected neural networks. If the model is split horizontally, the device which computes layer two has to wait for the messages with layer ones' activations. Therefore it is not possible that both devices can work in parallel. Vertical splitting is better, as the parts can work in parallel. An evaluation of the partitioning and scheduling problem is done by [9].

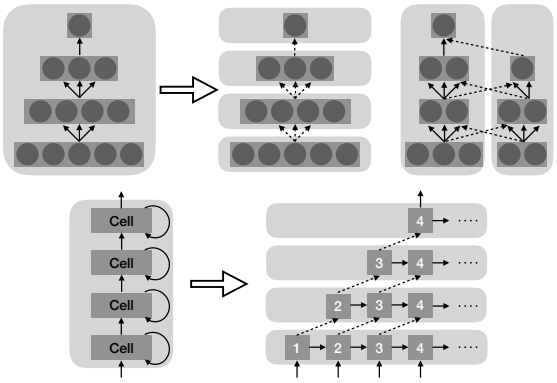

Figure 1: Splitting possibilities of a fully connected NN (top) and a RNN (bottom) for model parallelism.

On the contrary to this, partially connected networks like convolutional neural networks (*CNN*) can be split in a way that potentially speeds up the training time, given by their architecture.
Also, because recurrent neural networks (*RNN*) will unfold through time for training to eliminate their cycled structure, shown on the right side of figure 1, it is possible to train horizontally split RNNs efficiently. The computation of complex layers in parallel often outweighs the communication penalty.

2.2 Data Parallelism

Data parallel algorithms, in general, uses multiple workers which process on a subset of the whole data set to scale computation [5]. For deep learning, the NN is replicated on each device and run the same training steps with different data in parallel. If the neural network fits into memory, there are two different advantages over model parallelism. It is

(1) independent of network architecture and has the
(2) possibility to hide communication costs.

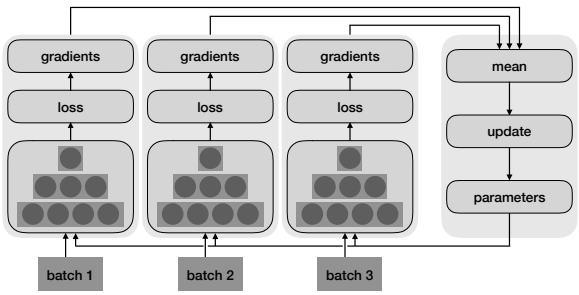

Figure 2: Example of data parallel training.

Figure 2 shows the basic functionality of training with data parallelism. One or more nodes, the parameter server (*PS*), is responsible for calculating the parameter updates and, if requested, redistribute them [7]. The updating step can be done in basically two different ways *synchronous* and *asynchronous*, which has a high impact on training performance.

Synchronous updates: The PS waits for all messages from workers before computing parameter updates.

Asynchronous updates: For each received message the PS computes parameter updates.

Both ways have different advantages and disadvantages. The risk of PS bandwidth saturation is higher for synchronous updates because the gradient updates are sent and received almost at the same time. Another issue is the de facto implemented barrier, where all workers synchronize and have to wait for the slowest. As a result, synchronous training usually achieves a lower throughput. [2]

In contrast, because there is no waiting time, training with asynchronous updates can achieve higher throughput. However, computed gradients could be outdated because while a worker computes gradients the PS updated model parameters. These are called *stale gradients* and can hurt training performance regarding training speed or possible accuracy. [2]

3 COMPARISON OF FRAMEWORKS

In the past decade, a lot of different deep learning frameworks have emerged. Most of them were developed by companies and released later under an open source license. Others have scientific origins and were developed by universities. In this section, we give an overview of different distinguishing properties of deep learning frameworks and describe *TensorFlow* and *MXNet* in more detail.

3.1 Static and Dynamic Graphs

Differences between *static* and *dynamic* Frameworks include the way how they implement networks. Static frameworks, like *TensorFlow* [1], are using two phases; the construction phase and the execution phase. In the construction phase the computation graph is created exclusively and will only be run in the execution phase. This approach has the advantage that the operations could be more optimized, for the given environment regarding the distributed training and operations which are used. However, there are some

issues with dynamic input data for example with different input image sizes or most tasks of natural language processing (*NLP*), where it is common that sentences consist of different length. [10] So there is the need to define a maximum input length and pad the rest, which leads to performance drawbacks.

Dynamic Frameworks like *PyTorch* are better in these use cases because of there natural dynamic characteristics. However, their optimization strategies are not as good as for static graphs. Because at the point of time when optimization is done only a partial graph is known [11].

3.2 Type of Distributed Training

As mentioned above, there are different types of distributed training: model and data parallelism. If synchronous updates for data parallelism are used, it is common to use a master-slave pattern. One of the workers, the master, is responsible for computing model updates and holding parameters. Others, the slaves, compute the gradients. For example, Microsoft Cognitive Toolkit implements this synchronous approach [3].

If asynchronous updates are used, there is a need for a PS approach which can be implemented in different ways. The above described centralized implementation is implemented by most frameworks, e.g., TensorFlow [1], Deeplearning4j [14], and CNTK [3]. Also, a decentralized approach exists, it is less common and is implemented by MXNet [4].

3.3 Exemplary Frameworks

In the following sections, we give an overview of exemplary frameworks, which fit our goal to evaluate the centralized and decentralized parameter server approach experimentally. We choose the most popular frameworks for the centralized approach, *TensorFlow* and the only available with a decentralized implementation of PS, *MXNet* [16].

3.3.1 TensorFlow. This deep learning framework was initially implemented by Google Brain[5] in 2011 and open sourced under the Apache 2.0 license in November 2015. As mentioned above it uses a static computation graph and implements the centralized Parameter Server concept [1].

Figure 3 shows distributed data parallel training with two workers and one PS. After the workers have calculated the gradients, they use the `push` command of PS interface to send them. With the blocking command `pull`, they wait for the correspondingly updated parameters computed by the PS [7].

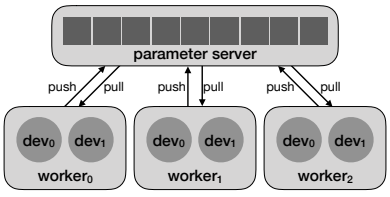

Figure 3: Example with Parameter Server.

[5]Deep learning research team at Google: https://ai.google/research/teams/brain

3.3.2 MXNet. The deep learning framework was implemented by DMLC (Distributed (Deep) Machine Learning Community)[6] and has been an Apache Incubator project since the beginning of 2017. It is trying to combine the advantages of *declarative* programming, which leads to a static computation graph, and *imperative* programming, which produce a dynamic graph for instance for easier debugging or other use cases in general. [4]

Figure 4 shows its decentralized concept for distributed data parallel training, in contrast to TensorFlow, it implements the PS concept by using a distributed *KVStore*. It also provides the same Interface (`push`, `pull`) as PS, but uses a two-level structure. On each worker machine, the KVStore first manage synchronization of this machines devices and second synchronize with other KVStores [4].

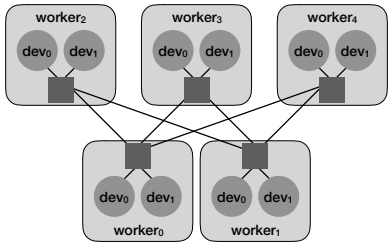

Figure 4: Example with distributed KVStore.

4 EXPERIMENTAL EVALUATION OF DISTRIBUTED TRAINING PERFORMANCE

Because of the importance of data parallel training in distributed settings, we aim to show the performance and scalability of *asynchronous data parallel* training with multiple worker nodes. Our goal is to compare the centralized and decentralized PS approaches experimentally. For this we use two neural networks, a CNN to represent smaller and easier to compute networks and a RNN which serves as a more complex case.

To visualize results, training time with different amounts of workers was measured and the corresponding throughput in *training samples per second* was calculated. We replicated the whole data set on each worker node, the corresponding calculation for n workers shows equation 1.

$$throughput_n = \frac{no.\ examples * epochs * no.\ workers}{training\ time_n} \tag{1}$$

To give information about the scalability of experiments we calculated the *speedup* (equation 2), which indicates how much the throughput with n workers could be increased.

$$speedup_n = \frac{throughput_n}{throughput_1} \tag{2}$$

With each configuration we did three experiments of two epochs training to make sure there are no outliers. For experiments, the most common frameworks which are implementing these concepts were used: *MXNet* and *TensorFlow*.

[6]Group to collaborate on open-source machine learning projects: http://dmlc.ml

Another interesting point is the usability of the frameworks. Because MXNet's distributed KVStore is part of each worker it scales up with the number of workers, therefore no further work is needed to find the best configuration. In contrast, the number of parameter server for TensorFlow is variable and can affect its performance. However, to find the best configuration can be hard. Therefore, to get similarly low effort for the training setup, we used for each TensorFlow experiment one dedicated parameter server.

In this section, we first specify the environment of evaluation experiments, describe the neural networks and used technology stack. In the second part, we show different experiments with asynchronous data parallelism with TensorFlow and MXNet. Third, we discuss the results.

4.1 Experimental Setup

For the experimental evaluation to show the scalability of the two concepts centralized and decentralized PS for distributed asynchronous data parallel training, we decided to choose a state-of-the-art Kubernetes-based cloud technology stack. Because for most companies or researchers in the area of deep learning it is cheaper to use on-demand cloud infrastructure, as buying and managing their hardware, especially with small teams and occasional use.

We used Googles' managed Kubernetes Service[7] for running experiments. Up to 25 worker nodes of machine type *n1-standard-1* were used. These consists of Intel Xeon E5 (Sandy Bridge) CPUs which has a Base Frequency of 2.6 GHz.

To manage Kubernetes resources, we are used the so-called package manager *Helm*[8]. Helm allows to easily start, stop and scale the training of test neural networks horizontally. The main feature is its template engine which is used to dynamically produce a set of worker nodes dependent on the value of a variable or argument.

An alternative for our implementation with Helm is *Kubeflow*[9]. Kubeflow is a collection of tools to simplify the deployment of machine learning workflows. Therefore it is not as lightweight and at the time of this paper, it does not support MXNet for training. These drawbacks are the reason for choosing our implementation.

To run experiments, a Docker container which can execute Python training scripts is required. We used the Ubuntu 16.04[10] base image and built a TensorFlow container version 1.8.0 and Python 3.5.2, as well as one for MXNet version 1.3.0 and Python 2.7.12.

To simulate smaller neural networks which consist of a less complex architecture we implemented the five-layered CNN *LeNet-5* [6]. It is implemented to do a classification of input images in 10 exclusive classes. For training, we used Zalando's *Fashion MNIST* Dataset [15], because it is a replacement for the well-known MNIST dataset[11] it also consists of 70.000 images. Each example is a 28x28 pixel gray-scale image associated with one of 10 classes.

For experiments with more complex neural networks, we used the RNN introduced by Zaremba et al. [17]. Its goal is to predict the next word in a textual context, also known as language modeling. The RNN consists of two Long short-term memory (*LSTM*) layers with 200 units per block and was unrolled for 20 steps. The network was trained on the Penn Tree Bank (*PTB*) dataset [8] we downloaded from Tomas Mikolovs' webpage[12]. It consists of about 1.000.000 words of English sentences and a vocabulary of 10.000.

For both, CNN and RNN, the whole dataset is loaded into memory before training starts. Therefore there is no time needed for loading data from disk which must be considered.

4.2 Experiments and Results

The Goal of the experiments is to analyze the behavior of TensorFlow and MXNet and the corresponding concepts with an increasing number of worker nodes. To avoid an experimental setup with an extremely large amount of worker machines, an alternative approach was used. Training with a smaller batch size leads to higher synchronization effort, and therefore the frameworks scalability have to be better to prevent a performance drop.

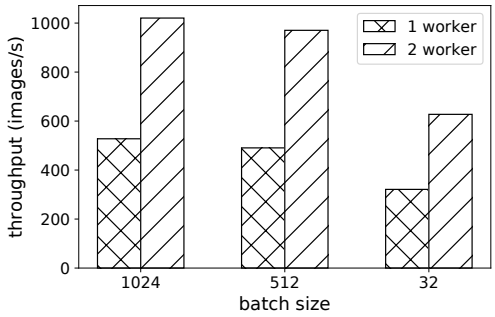

Figure 5: Comparison of MXNet experiments with different batch sizes and worker nodes.

To show the correctness of the above mentioned alternative approach, figure 5 compares the results of MXNet CNN experiments with 1024, 512 and 32 test images per batch with one and two workers. As expected, it shows that the increasing performance has about the same factor as the number of workers. Also, a slightly decreasing throughput at smaller batch sizes can be noticed.

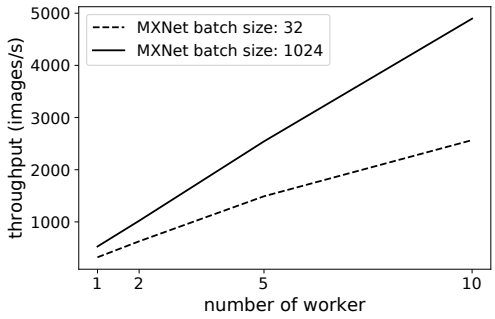

Figure 6: Demonstration of the alternative approach, the smaller batch size, the less efficient its scalability.

[7] Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine/
[8] Helm - The package manager for Kubernetes: https://helm.sh
[9] Kubeflow - The Machine Learning Toolkit for Kubernetes: https://www.kubeflow.org
[10] Docker Ubuntu Image: https://hub.docker.com/_/ubuntu/
[11] MNIST dataset: http://yann.lecun.com/exdb/mnist/
[12] http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

Figure 6 shows that the above mentioned approach works. With a batch size of 1024 images, MXNet can scale linearly up to 10 workers. However, if the batch size is much smaller the computation-communication ratio is lower, which leads to less efficiency in terms of the framework's throughput.

4.2.1 Convolutional NN. We did different experiments with an increasing number of worker nodes for both frameworks. Figure 7 shows the results.

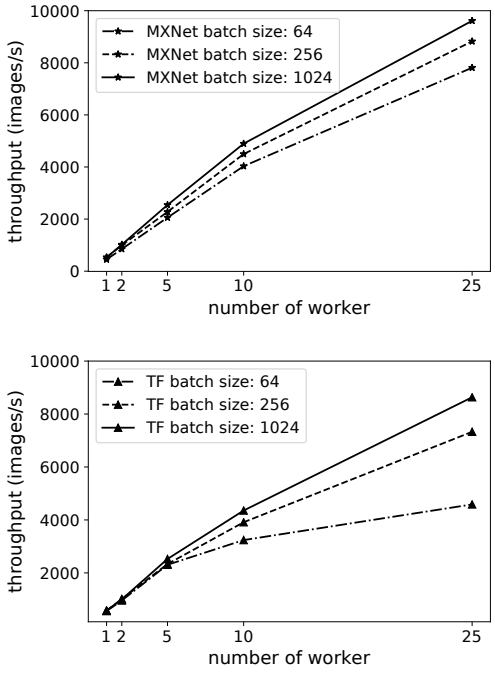

Figure 7: Experimental results for CNN with MXNet (top) and TensorFlow (bottom)

First, it is shown that MXNet can scale linearly up to 10 workers with the given configuration, in contrast to TensorFlow which only scales linearly up to 5 workers.

Second, with 10 or more workers MXNets' throughput is for each experiment higher than TensorFlows', especially for small batches. With a batch size of 64 images, TensorFlow could only achieve a scale up efficiency of about 32%. In contrast, MXNet achieves about 69%.

Figure 8 compares the results described above. It shows TensorFlows' and MXNets' speedup each with a batch size of 64 and 1024. Corresponding to MXNets' higher throughput, its speedup is higher too. The gap increases with an increasing number of worker nodes and smaller batches. With a batch size of 64 and 25 worker nodes, MXNets' speedup is more than two times higher as TensorFlows'. Another point to mention is the different performance drop from 1024 to 64 batches. For MXNet this is only about 5%. For TensorFlow it is about 45%.

4.2.2 Recurrent NN. Figure 9 and 10 show the results of RNN experiments with respect to throughput and speedup. Similar to the

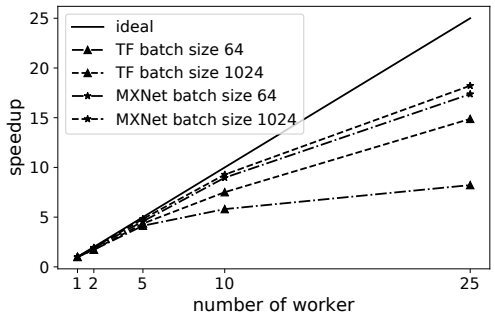

Figure 8: Comparison of speedup for CNN

CNN experiments both frameworks can scale up with bigger batch size more efficiently. However, in contrast for RNN MXNet achieves for each configuration higher throughput as figure 9 shows.

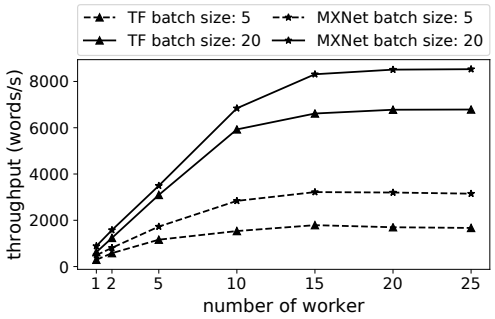

Figure 9: Experimental results for RNN

As mentioned, MXNets' throughput is higher, however as figure 10 shows for batch size 20 TensorFlows' speedup is better. It also shows the trend that with more workers MXNet can decrease this gap. For batch size 5 it is the other way around, the more workers are used the better is MXNets' speed in comparison to TensorFlow.

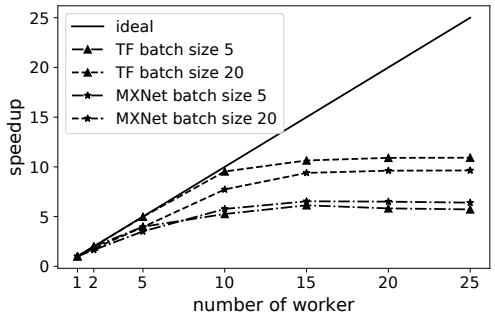

Figure 10: Comparison of speedup for RNN

For batch size 20 MXNet can achieve an efficiency of 77% with 10 workers and 62% with 15. For batch size 5 it is only 57% and

43%. In contrast, TensorFlow is much better with batch size 20 its efficiency is 95% and 70%, but for batch size 5 it only can achieve an efficiency of 52% and 40%. However, consider MXNet's higher throughput. For both, it is not practical to use more worker nodes.

To summarize the experiments, for the CNN experiments TensorFlows' throughput with few workers (1 to 5) is slightly higher than MXNets'. For increasing batch sizes this gap gets smaller. This trend, the more complex or bigger regarding calculations the NN becomes, the higher MXNet throughput becomes, could also be validated with the RNN experiments. There, MXNet could achieve for each configuration higher throughput and with bigger batch sizes the distance to TensorFlow increases.

4.3 Discussion

Only in a few cases, with few worker nodes and less complex NN, it is preferable regarding training performance to use TensorFlow. Otherwise, it is better to use MXNets' distributed KVStore, especially with 5 and more worker nodes or more complex NN. To be fair, TensorFlows' PS could be explicitly scaled which has to be implemented and is out of scope in this paper.

MXNets' approach does not only scale better in our training environment, but it also requires less and, above all, less complicated code. For instance, there is no need to implement different training scripts or, at least, a generic script and use different parameterization for each worker.

For MXNet training is conducted by a so-called scheduler, which starts all worker processes on nodes and initializes them. E.g. it assigns worker numbers and distributes IP-addresses to them. In contrast, TensorFlow requires to start its worker process for each node. In an automated environment, like our Kubernetes- and Helm-based technology stack, both approaches are good. However, without any automation MXNet is more convenient.

To wrap up our experiences, MXNet is a good choice, especially when dealing with more complex NN. It showed better scalability and mostly higher throughput. Moreover we found the code to be less complicated and the configuration to be easier.

5 RELATED WORK

Experiments with a distributed configuration based on GPUs with up to 4 nodes were done by Zhu et al. [18]. They focused on creating a new benchmark to analyze frameworks on the major deep learning domains. However, similar to our results, they showed MXNets' higher throughput with ResNet-50, as well as the increased performance with bigger batches.

Another benchmark evaluated distributed training on single machine configurations with CPU, multiple CPUs, GPU and multiple GPUs [13]. Their GPU and multi GPU results are similar to ours. They found that MXNets' scalability in the most cases beats TensorFlows'. Their CPU-based training results show, in contrast to ours, for less complex NN that MXNet can achieve better performance than TensorFlow. For more complex NN, like ResNet-50, it is the other way around. A possible reason might relate to their older framework versions (TensorFlow: 0.11, MXNet: 0.7.0).

In contrast to our focus on asynchronous data parallel training a benchmark by Shi et al. aimed at synchronous data parallel training [12].

All three studies above [12, 13, 18] focus on bare metal setups. We used a more practical approach with a state-of-the-art Kubernetes-based cloud technology stack.

6 CONCLUSION

In this paper, we compared the two common concepts for asynchronous data parallel training, *Parameter Server* and *distributed KVStore*. For the experimental evaluation, we used a state-of-the-art Kubernetes-based cloud technology stack with up to 25 workers of the machine type n1-standard-1 of Googles' Kubernetes Engine. For experiments, the well-known frameworks TensorFlow and MXNet were used to measure their throughput with an implementation of a CNN and RNN. Our experimental results show that especially for a low computation-communication ratio MXNet can achieve higher throughput and therefore mostly better speedup.

For future work we plan experiments with an optimized TensorFlow setup regarding the number of used parameter server. Second, it would be interesting to see their performance if GPU cluster were used. Moreover, we plan to use further state-of-the-art neural networks.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, et al. 2016. Tensorflow: a system for large-scale machine learning. In *OSDI*, Vol. 16. 265–283.

[2] Jianmin Chen, Xinghao Pan, Rajat Monga, et al. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[3] Kai Chen and Qiang Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 5880–5884.

[4] Tianqi Chen, Mu Li, Yutian Li, et al. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[5] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[7] Mu Li, David G Andersen, Jun Woo Park, et al. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, Vol. 14. 583–598.

[8] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics* 19, 2 (1993), 313–330.

[9] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The Tensorflow Partitioning and Scheduling Problem: It's the Critical Path!. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning (DIDL '17)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3154842.3154843

[10] Graham Neubig, Chris Dyer, Yoav Goldberg, et al. 2017. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[11] Adam Paszke, Sam Gross, Soumith Chintala, et al. 2017. Automatic differentiation in pytorch. (2017).

[12] Shaohuai Shi and Xiaowen Chu. 2017. Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs. *arXiv preprint arXiv:1711.05979* (2017).

[13] Shaohuai Shi, Qiang Wang, Pengfei Xu, et al. 2016. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*. IEEE, 99–104.

[14] Deeplearning4j Development Team. 2018. Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0. http://deeplearning4j.org

[15] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[16] Jan Zacharias, Michael Barz, and Daniel Sonntag. 2018. A Survey on Deep Learning Toolkits and Libraries for Intelligent User Interfaces. *arXiv preprint arXiv:1803.04818* (2018).

[17] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[18] Hongyu Zhu, Bojian Zheng, Bianca Schroeder, et al. 2018. DNN-Train: Benchmarking and Analyzing DNN Training. In *SysML 2018*.