Received: 22 February 2021 | Revised: 6 July 2021 | Accepted: 2 November 2021

DOI: 10.1002/cpe.6730

SPECIAL ISSUE PAPER

WILEY

Single node deep learning frameworks: Comparative study and CPU/GPU performance analysis

Jean-Sébastien Lerat[1,2] | Sidi Ahmed Mahmoudi[1] | Saïd Mahmoudi[1]

[1]Faculty of Engineering, University of Mons, Mons, Belgium

[2]Department of Sciences and Technologies, Haute École en Hainaut, Mons, Belgium

Correspondence
Jean-Sébastien Lerat, Faculty of Engineering, University of Mons, 20 Place du Parc, 7000 Mons, Belgium.
Email: Jean-Sebastien.Lerat@umons.ac.be

Abstract

Deep learning presents an efficient set of methods that allow learning from massive volumes of data using complex deep neural networks. To facilitate the design and implementation of algorithms, deep learning frameworks provide a high-level programming interface. Based on these frameworks, new models, and applications are able to make better and better predictions. One type of deep learning application is the Internet of Things that can gather a continuous flow of data, which causes an explosion of the amount of data. Therefore, to handle this data management issue, computation technologies can offer new perspectives to analyze more data with more complex models. In this context, a cluster of computers can operate to quickly deliver a model or to enable the design of a complex neural network spread among computers. An alternative is to distribute a deep learning task with HPC cloud computing resources and to scale cluster in order to quickly and efficiently train a neural network. As a first step to design an infrastructure aware framework which is able to scale the computing nodes, this work aims to review and analyze the state-of-the-art frameworks by collecting device utilization data during the training task. We gather information about the CPU, RAM and the GPU utilization on deep learning algorithms with and without multi-threading. The behavior of each framework is discussed and analyzed in order to shed light on the strengths and weaknesses of the different deep learning frameworks.

KEYWORDS

artificial intelligence, CPU, deep learning, distributed computing, frameworks, GPU, parallel computing

1 | INTRODUCTION

Over the last decade, deep learning has been intensively used due to the explosion of data. Enterprises accumulate data in order to form a strong and representative dataset that they wish to exploit. Moreover, recent technologies, like the Internet of Things devices, frequently gather information about their environment and communicate to centralize them in order data. Such a large amount of data is called big data. Traditional machine learning models are efficient but cannot handle this increasing amount of data because of the complexity of algorithms and the high number of features hidden in the data. On the other hand, the recent advances in hardware, especially in graphics processing units (GPU), have made neural networks able to efficiently process such data and reduce the computing time. The explosion of data, as far as the computation capabilities are concerned, offer new perspectives to analyze data through more complex models. These models represent the main component of the deep learning domain, which is a growing trend for both scientific research and enterprises that want to understand their data or to automate specific tasks, such as image classification, object recognition, facial recognition, and so forth. The increase in data also occurs on high-definition images and videos requiring alternative processing of this significant amount of data. Such alternatives are distributed computing and cloud computing, well-known and developed fields.

The research area in distributed deep learning focuses on how to reduce network communication because it is an issue for large scale models. Different approaches have been considered to decrease the network communication, such as gradient quantization[1] or the gradient sparcification.[2,3] There are other methods, such as fast memory access,[4] which is used to synchronize the gradient among computing nodes. Frameworks have been developed or extended[5,6] to implement these sets of methods but none of them discuss how the framework performs on a single node compared to the state-of-the-art. As a first step to design a framework that automatically scales on an HPC cloud computing infrastructure, we first analyze and compare current deep learning frameworks.

Response time and computation capabilities are a common concern of deep learning practitioners. A usual solution to this problem is to distribute the deep learning task on a cluster. In 2015, a Spark interface was proposed[7] to facilitate distributed deep learning, which increases computation capability and allows the response time to be decreased. In Reference 8, the speedup of four deep learning frameworks was analyzed from a single GPU to a multi-GPU environment. The best combination of distributed deep learning was Caffe-MPI, which outperformed MXNet, Tensorflow, and CNTK. Nevertheless, these comparisons are outdated due to the evolution of the frameworks. For instance, the Caffe framework evolved and has been integrated into the PyTorch framework. Moreover, the analysis is only based on the distribution, without taking into account local considerations. This is how frameworks behave on a single node. In fact, they only use the single node as a baseline to measure the speedup of distributed deep learning. Nevertheless, deporting computation to the edge implies small computation capabilities. A recent work[9] compared frameworks in terms of API, support for distributed and parallel deep learning and community, without taking into account their performances. From their comparison, Tensorflow is the favorite framework of the community, which contradicts performance comparisons. In our article, we focus on a single node with a single device in order to compare frameworks on their local performance.

2 | METHODS

The purpose of this article is to evaluate deep learning frameworks based on their execution time on both the CPU and the GPU, as well as to understand the difference of their behavior. These frameworks are compared on the basis of their stability. To achieve this goal, we have compared different deep neural network architectures on two datasets in order to observe the impact of the model complexity and the amount of data.

2.1 | Framework overview and selection

Due to the numerous frameworks, the focus is on a subset of frameworks that meet the requirements needed for quick response time and proper use of resources. These requirements are:

- A native implementation allowing a response to the need for performances. Such implementation removes the overlay of programming layers, such as the interpreter.
- CUDA and OpenCL support that enables the use of GPUs for computation.
- A Python interface for quick prototyping in a language commonly employed within deep learning communities, in terms of use and support. Python is a script language with concise syntax that enables fast implementation and is widely used by the community.
- Active development to ensure continuous development and integration of new technologies.

The subset that we propose to investigate in this article is composed of MXNet, Paddle, pyTorch, SINGA, and TensorFlow, which are initiated or supported by the Apache Foundation, Baidu Incorporated Companies, Facebook Incorporated Companies, the National University of Singapore, and Google Limited Liability Companies, respectively.

2.2 | Deep neural networks

The evaluation of frameworks requires the training of deep neural networks that differ in complexity, either based on the number of layers and neurons per layer or on the neural complexity (density and sparsity). Without loss of generality on the type of neural networks, the current work focuses on convolutional neural networks (CNN) which are well adapted for image and video classification problems. These networks are well known networks, used, for example, by the community for the ImageNet Large Scale Visual Recognition Challenge.

In order to provide a fair analysis, we selected four accurate classification architectures, which are based on deep neural networks with different levels of complexity:

AlexNet,[10] VGG 16,[11] ResNet 50 v1,[12] and MobileNet v2 x1.0[13] because they differ in terms of layers and neural complexity. The last layer of these networks corresponds to the number of classes. In this work, it is adapted to three classes.

In 2012, the AlexNet[10] CNN architecture won the ImageNet competition and outperformed all the prior competitors. It was a more complex model than the state-of-the-art at that time, allowing the top five errors to be significantly reduced from 26% to 15.3%. This architecture was stacking convolutional layers—11×11, 5×5 and 3×3—with max pooling, ReLu activation function and using dropout. The learning process was a stochastic gradient with momentum. The authors were able to train their model due to an efficient implementation on two GPUs.

In 2014 the Visual Geometry Group[11] proposed new CNN architecture in the same ILSVRC competition as AlexNet. This group gave the name VGGNet to their architecture, which is often referred as only VGG only. The original version was composed of 16 convolutional layers—138 million parameters—and is referenced under the name VGG16.

In 2015, the new residual neural network CNN architecture,[12] simply called ResNet, was proposed for the challenge on the ImageNet dataset. This architecture was not only adaptable via a hyper-parameter customizing its number of layers, but also proposed a new concept: residual learning. Usual CNN architectures alternate between convolutional layers and pooling layers in order to extract features from images. Residual learning is designed to learn some residual features that can be viewed as a subtraction of learning features. ResNet implemented residual learning through shortcut connections, using the output of layer i as an additional input of layer j with $j > i + 1$. This kind of connection also helps the network to avoid the phenomenon of vanishing gradients, which occurs when several layers, between which the gradient computation becomes smaller and smaller, do not update the model.

The MobileNet version 2[13] can be viewed as a simplified version of ResNet. It also uses shortcut connections but requires fewer parameters than ResNet. This happens because it is built from the first version of MobileNet and uses depthwise separable convolution instead of bottleneck blocks. Instead of a convolution layer, MobileNet uses a depthwise separable convolution which requires less computation. It composed of two sub-layers:

- a depthwise convolution layer, only applying a 2D convolution layer to each image channel;
- a pointwise convolution layer, applying a stride of 1×1 to the three channels.

The second version of MobileNet uses an inverted residual convolution layer, an extension of the depthwise separable convolution. This extension presents three modifications:

1. The use of shortcut connection.
2. The application of a 1×1 convolution layer to expand data before applying a depthwise separable convolution layer. This expansion creates new channels, the number of which depends on a hyper-parameter called the *expansion factor*.
3. The application of projection to channels so that only one of them remains after the depthwise separable convolution step.

Unlike other architectures, MobileNet does not use pooling layer but sets up strides in order to decrease the amount of data.

2.3 | Training task

The training task of the image classification problem has to take care of how to feed the neural network with images. The latter have to be preprocessed in order to fit the input required by neural networks. In this work, such pre-processing is designed based on the original publication of the selected convolutional neural networks instead of designing the most accurate model. This is why we pre-process input images to a 224×224 with the 3 RGB channels. The goal is to measure and quantify the resource usage of a common learning task. The image pre-processing pipeline follows the sequences:

1. Image crop/scaling to 224×224
2. Random horizontal flip transformation
3. RGB-normalization with $\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$
4. Conversion to tensor data structure

The optimizer is the stochastic gradient with a learning rate $\alpha = 0.001$ and a momentum $\mu = 0.9$. The loss is computed with the cross-entropy method.

The Computer Sciences Department of the Faculty of Engineering of the University of Mons has collected and provided a dataset for the current work. This dataset has a small version composed of 791 photos and a big version composed of 6003 photos. Each of these are split into three distinct classes: fire, smoke, and no fire. This dataset is used to generate a deep learning model for fire or smoke detection from images.

2.4 | Setup

To measure the resource use, setups are designed such that frameworks can exploit either CPU or GPU. Then the learning task is adapted to train in mini-batch mode to analyze how the frameworks adapt their behavior when they can simultaneously process data and load the next data. These setups are:

1. A single CPU core—without multi-threading—for image processing and training.
2. A single GPU automatically fed with data by the framework to train the network.
3. Setup 2 where data are processed in a mini-batch of size 100.

3 | RESULTS

The training was done on a computer with an `AMD 2950X RYZEN THREADRIPPER CPU`and a `EVGA GeForce RTX 2080 Ti GPU`. Hardware and software details are reported in Appendix A.

In Setup 1, all the frameworks were configured to process the training on a single CPU using a single thread. They were able to manage the small and the big datasets, except Singa, which uses a thread per image. This behavior led to allocation errors on the big dataset. The average times over 100 epochs are reported in Table 1. As shown, the biggest dataset required more time because it is composed of more images on all frameworks. The different CNN architectures required more time due to the number of neurons and the number of layers. On Paddle, pyTorch, and Singa the architectures from the least to the most greedy are AlexNet, MobileNet, ResNet, and VGG. For other frameworks, AlexNet and MobileNet are exchanged. Singa was not able to load the big dataset properly because it created a thread per image which led to an allocation error. Additional codes are required to bypass this limitation which is a weakness for a high-level programming framework. Nevertheless, Singa is the newest framework and will be improved over time. On average, the fastest framework on the CPU was pyTorch which outperformed all the others. The only use case where pyTorch was outperformed, by 40 s, was by MxNet on the MobileNet v2 x1.0 architecture.

In Setup 2, the training was transferred to the GPU. Table 2 lists the average time per epoch for each framework. They all gained in time, except Paddle, on the big dataset. Clearly the best results were on Singa, which took 6 s for all architectures on the small dataset. However, Singa was not able to manage the big dataset without additional codes to restrain its number of threads during disk access. Unlike Singa, MxNet could manage both datasets and came second, followed closely by pyTorch. Paddle and Tensorflow performed less well because they required more than 10 times the time to achieve the same goal.

In order to understand the difference between frameworks when using the GPU, we inferred the behavior with the help of the NVidia—designer of the graphic card—tool that logs the time spent in each CUDA function. CUDA is the API offered by NVidia to exploit their GPU, which is used by

TABLE 1 Average time in seconds per epoch for small|big datasets on Setup 1 (CPU)

Frameworks	AlexNet	MobileNet v2 x1.0	ResNet50	VGG16
MxNet	401|2993	181|1391	547|4181	2178|16,480
Paddle	430|1107	431|1045	2273|5790	7273|16,493
pyTorch	139|879	221|1459	480|3122	630|9010
Singa	273|None	430|None	977|None	1227|None
TensorFlow	528|3957	420|3223	835|6289	1979|16,397

TABLE 2 Average time in seconds per epoch for small|big datasets on Setup 2 (GPU)

Frameworks	AlexNet	MobileNet v2 x1.0	ResNet50	VGG16
MxNet	19|114	28|182	33|232	28|185
Paddle	328|2380	338|2533	332|2500	337|2462
pyTorch	23|122	40|254	43|271	34|190
Singa	6|None	6|None	6|None	6|None
TensorFlow	343|2542	341|2564	351|2605	350|2648

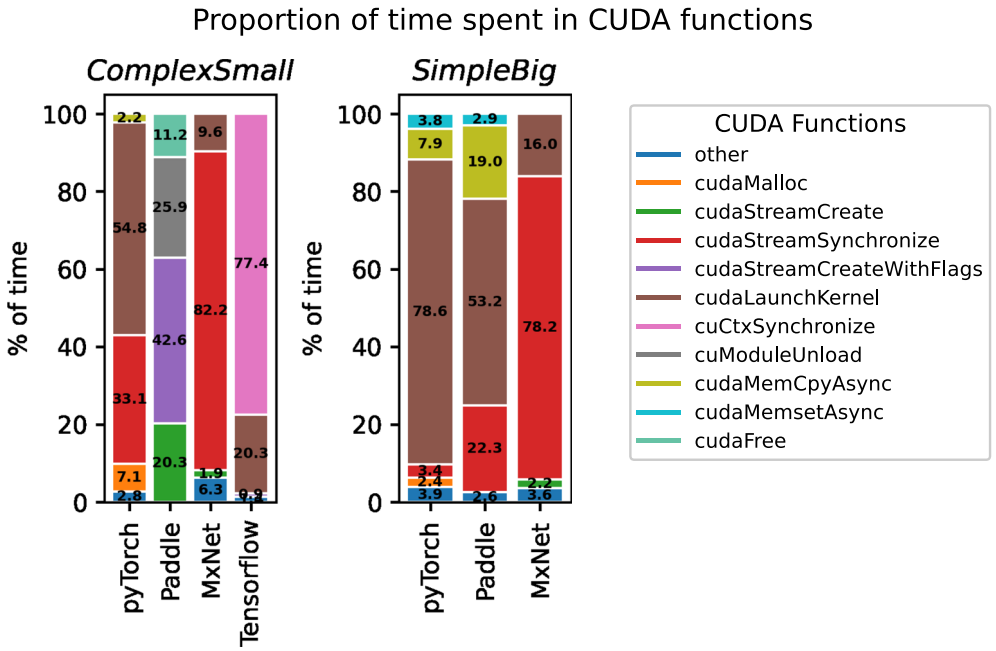

FIGURE 1 Percentage of time spent in CUDA functions for each combination of frameworks and datasets on Setup 2. The doughnut from the center to the outside, is for Tensorflow, MxNet, Paddle, and pyTorch, respectively

all the frameworks. In the previous results, presented in Tables 1 and 2, we have shown the complexity of the CNN architecture and the dataset. To exhibit the differences of behaviors we focus on two use cases:

1. SimpleBig: AlexNet on the big dataset which is a less complex model on numerous images.
2. ComplexSmall: VGG16 on the small dataset which is a more complex model on few images.

The computational complexity of these use cases lies either in the amount of data or in the complexity of the model. Consequently, they allow us to observe and understand how frameworks behave in these extreme cases.

Singa is a promising framework but is not mature enough to handle the amount of data. This is why we removed it from our analysis. The percentage of time spent in the top-10 functions is illustrated in the stacked bar chart in Figure 1. The SimpleBig use case can be processed quickly by the GPU because the model complexity is lower because it has fewer layers. This is why the time spent by the framework to train the model highly depends on how the framework feeds the GPU with data. MxNet creates a stream of data and spends most of its time synchronizing the streams, contrary to Paddle and pyTorch which mostly call cudaLaunchKernel which applies custom functions to blocks of data. The main difference between them is that pyTorch spent 25.3% of its time in this function and synchronized its streams less. We know that synchronization has a cost because of the idle time of other threads. Tensorflow could not be analyzed on the SimpleBig use due to the insufficient disk space. Tensorflow processed slowly. Consequently, as the amount of data became bigger, the NVidia tools could collect data for a such a long period of time on 1 TB of free space. To overcome this limitation, we will investigate a multi-node infrastructure in a further work. For the ComplexSmall use case, pyTorch behaved similarly except that it needed more synchronization. The time spent in cudaStreamSynchronize was 29.7% higher. Moreover, it also allocated memory more often. Indeed, this use case required better GPU management because of the model's complexity. The behavior of MxNet was quite similar to the SimpleBig use case. Paddle completely changed its behavior, spending 62.9% of its time by creating streams with and without flags, which decreased the number of synchronizations. However, the cost of Paddle comes from the 31.1% of its time spent to free the GPU memory via cudaFree and cuModuleUnload. Tensorflow spent double the time that MxNet did in cudaLaunchKernel but 77.4% of its time to synchronize contexts.

The CUDA behavior of the frameworks depends on the number of operations processed on the GPU and how to feed the GPU with the data. The two CNN use cases mainly differ in the number of layers and the size of the layers.

Training on the GPU is a factor that influences both the results and how the framework feeds the GPU with data, especially for the SimpleBig use case. This is confirmed by the impressive time per epoch result on Singa. The CPU is in charge of loading images and transferring data to the model located on the GPU. To understand the difference in behaviors between frameworks on how they load images, we analyzed the CPU, the RAM and the number of threads illustrated over execution time in Figure 2. All the frameworks quickly increased to the full CPU utilization, except Paddle on the SimpleBig use case and Tensorflow on the ComplexSmall use case. In this case, the CPU was underused. Paddle and Tensorflow also progressively

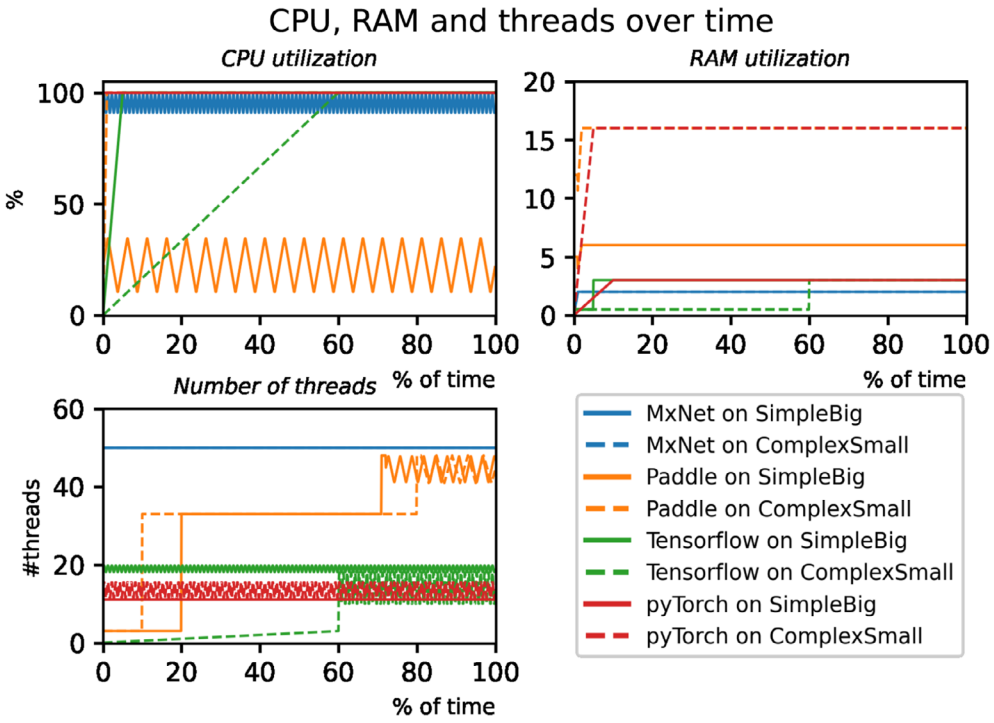

FIGURE 2 CPU, RAM and number of threads used by frameworks on the two use cases: ComplexSmall and SimpleBig, Setup 2

increased the number of threads, and these oscillated, unlike with other frameworks. The exception was pyTorch on the ComplexSmall use case. At the end of the computation and data loading, different thresholds were achieved: 50 on MxNet, between 12 and 18 on pyTorch, between 40 and 48 on Paddle, and between 10 and 20 on Tensorflow. The behavior of the used RAM followed the same behavior for all setups and frameworks, it quickly increased to a threshold. This threshold varied depending on the framework and the use case, except for MxNet, which had the same threshold for both. Paddle and pyTorch required more than 10% of memory on the ComplexSmall use case, unlike Tensorflow, which only required 3% of RAM for the SimpleBig use case and 1% for the ComplexSmall use case. The Tensorflow strategy on the CPU seemed less efficient than MxNet and pyTorch, but its memory requirement was close to MxNet and even better on the ComplexSmall use case.

The CNN architectures were analyzed as a whole in order to understand how they behave when layers are combined. That is the case when they are used in practice. Analyzing a specific kind of layer requires creating CNN architecture only composed of that layer. This limitation does not allow the understanding how a specific layer processes inside a CNN architecture. Moreover, a specific layer approach does not allow interactions between layers to be highlighted, which can be a place where frameworks save time.

Training deep learning models is usually done in a mini-batch fashion. That is why data are split into groups and the model is trained on each group sequentially. This was done in Setup 3, where we arbitrarily fixed the batch size to 100. The Paddle and MxNet frameworks were removed because they were unable to process in this configuration on our hardware, meaning that the frameworks produced allocation errors. The GPU behavior is not reported because the results were close to those presented in Figure 1. However, the CPU utilization changed, as illustrated in Figure 3. Tensorflow had the same behavior as in batch processing, but it took more time to achieve 100% of CPU utilization. The memory used by pyTorch was its weak point in Setup 2 on the GPU. In mini-batch, the amount of required memory was deceased to the same threshold as Tensorflow. As its number of threads no longer depended on the use case, it became constant.

4 | DISCUSSION AND CONCLUSION

During this work, we compared five deep learning frameworks in terms of resources use and response times on four CNN architectures trained on a small and a big dataset. We designed different setups to understand how frameworks perform and behave on both the CPU and the GPU. From our analysis, we have shown that Singa was unable to process a big dataset without additional code because it requires a thread per input data; Paddle and MXNet misbehaved when the mini-batch was used due to allocation errors. Tensorflow and pyTorch were the most stable frameworks in our setups.

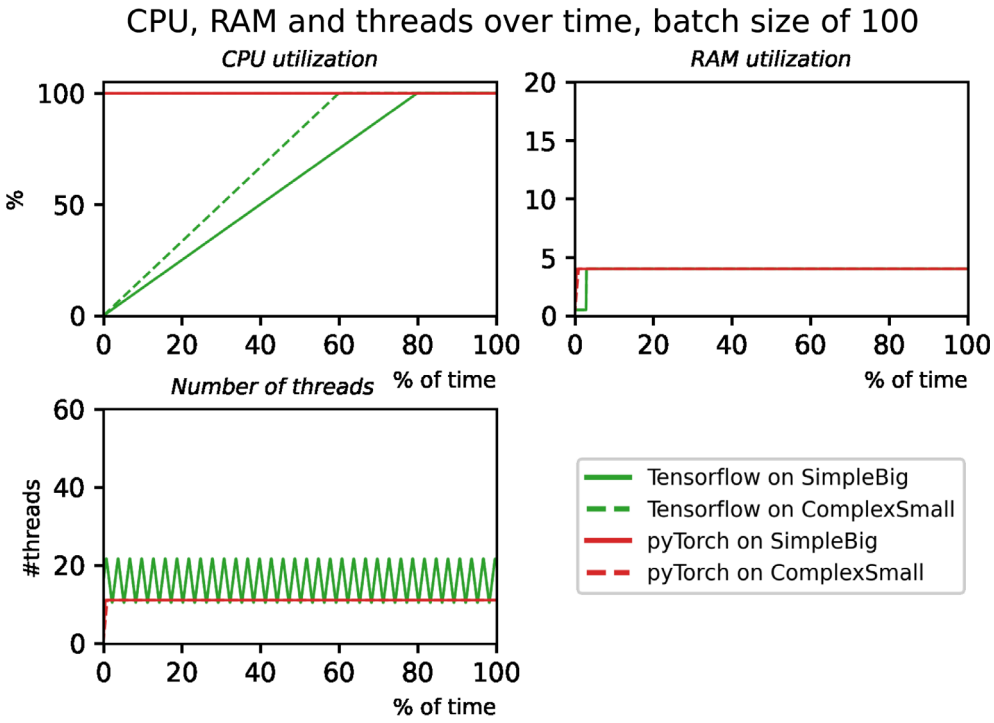

FIGURE 3 CPU, RAM and number of threads used by frameworks on the two use cases: ComplexSmall and SimpleBig, Setup 3

In almost all cases pyTorch outperformed other frameworks on the CPU, and it also outperformed Paddle and Tensorflow on the GPU. The drawback of pyTorch is that its memory requirements are much higher than others, like Tensorflow. Nevertheless, when it comes to the mini-batch mode, pyTorch drastically reduces its needs in memory and outperforms Tensorflow for the same cost of resources. To understand this difference, we analyzed the proportion of time spent in the NVIDIA functions. Tensorflow seemed to synchronize more, which had a higher time cost.

As discussed, pyTorch seems to be the best compromise between stability and performance on a single node. Nevertheless, it is known that parallel computing can speed up algorithms. Deep learning tasks can also be paralleled by training the model on distinct mini-batches while synchronizing the gradient. On the other hand, the cost of synchronization can impact the speedup such that parallel computing becomes slower, especially on the GPU, when data has to be loaded in its memory. In the same way, distributed computing is a field where fast response time is required. The benefits and drawbacks are equivalent but can be higher than is parallel computing. Each computing node can focus on a subset of input data, meaning that there is no need to load and unload input data in the memory, while gradient synchronization requires network messages that are slower than the synchronization of the RAM for the single node use case. In future work, we are interested in exploring how pyTorch behaves in these setups, as well as understanding when to use parallel computing, distributed computing, or no parallelism. This is quite substantial to understand, so that the edge computing architecture can be designed. Edge computing is a use case with a lot of devices with low computing capability that has to synchronize in a distributed computing fashion. We are also interested in exhibiting how frameworks process each distinct type of layer on the GPU, and especially which CUDA functions are used. Finally, we wish to design a framework that will automate the deployment of virtual machines or Docker containers in order to quickly obtain the results of a deep learning task. Best resource use and speedup are required in order to get a quick response time and to minimize the number of virtual machines, which reduces the cost.

ACKNOWLEDGMENT

This work was partially funded by the Wallonia-Brussels Federation (JCM/TP/BS/mo/c999)

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in GitHub at https://github.com/Belegkarnil/forestfire/releases/tag/1.0.

ORCID

Jean-Sébastien Lerat https://orcid.org/0000-0003-4468-0899
Sidi Ahmed Mahmoudi https://orcid.org/0000-0002-1530-9524
Saïd Mahmoudi https://orcid.org/0000-0001-8272-9425

REFERENCES

1. Wen W, Xu C, Yan F, et al. Terngrad: ternary gradients to reduce communication in distributed deep learning. Proceedings of the 2017 International Conference on Information and Communication Technology Convergence; 2017:1509-1519.
2. Sattler F, Wiedemann S, Müller KR, Samek W. Sparse binary compression: towards distributed deep learning with minimal communication. Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN); 2019:1-8; IEEE.
3. Kuang D, Chen M, Xiao D, Wu W. Entropy-based gradient compression for distributed deep learning. Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems; 2019:231-238; IEEE.
4. Lim EJ, Ahn SY, Choi W. Accelerating training of DNN in distributed machine learning system with shared memory. Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC); 2017:1209-1212; IEEE.
5. Li D, Lai Z, Ge K, Zhang Y, Zhang Z, Wang Q, Wang H. Hpdl: towards a general framework for high-performance distributed deep learning. Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems; 2019:1742-1753; IEEE.
6. Ahn S, Kim J, Lim E, Choi W, Mohaisen A, Kang S. Shmcaffe: a distributed deep learning platform with shared memory buffer for HPC architecture. Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems; 2018:1118-1128; IEEE.
7. Moritz P, Nishihara R, Stoica I, Jordan MI. Sparknet: training deep networks in spark; 2015. arXiv preprint arXiv:1511.06051.
8. Shi S, Wang Q, Chu X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. Proceedings of the 2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech); 2018:949-957; IEEE.
9. Mayer R, Jacobsen H-A. Scalable deep learning on distributed infrastructures: challenges, techniques, and tools. *ACM Comput Surv (CSUR)*. 2020;53(1):1-37.
10. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Adv Neural Inf Process Syst*. 2012; 25:1097-1105.
11. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Bengio Y, LeCun Y, eds. *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings, San Diego, CA*; arXiv; 2015.
12. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016:770–778.
13. Howard A, Zhmoginov A, Chen LC, Sandler M, Zhu M. Mobilenetv2: Inverted residuals and linear bottlenecks. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018:4510–4520.

How to cite this article: Lerat J-S, Mahmoudi SA, Mahmoudi S. Single node deep learning frameworks: Comparative study and CPU/GPU performance analysis. *Concurrency Computat Pract Exper*. 2023;35(14):e6730. doi: 10.1002/cpe.6730

APPENDIX A. HARDWARE AND SOFTWARE CONFIGURATION

The physical computer is composed of the following devices:

- **Graphic card** *EVGA GeForce RTX 2080 Ti*
- **Power supply** *Seasonic Prime 1300 W Gold*
- **Case** *Corsaire Obsidian 750D Airflow*
- **CPU** *AMD 2950X RYZEN THREADRIPPER*
- **CPU Cooling system** *Watercooling Enermax LIQTECH TR4 II 360 RGB*
- **Additional case FAN** *be quiet! Silent Wings 3 120 mm PWM High-Speed* with *SilverStone FF122*
- **Additional device** *DVD ASUS DRW-24D5MT*
- **Motherboard** *ASRock Fatal1ty X399 Professional Gaming*
- **RAM** *Corsair Vengeance LPX Series Low Profile 128 GB (8 × 16 GB) DDR4 2933 MHz*
- **Storage**

  Samsung SSD 970 PRO M.2 PCIe NVMe 1 To
  Samsung SSD 860 EVO 4 To

TABLE A1 Installed software

Python related setup	NVidia software
python 3.6.9	nsight-compute 2019.5.0
pip3 9.0.1	nsight-systems 2020.2.1
pbr 5.4.4	libcudnn7_7.6.5.32-1+cuda10.2
stevedore 1.32.0	cuda-repo-ubuntu1804_10.2.89-1
virtualenv-clone 0.5.3	tensorRT 7
virtualenvwrapper 4.8.4	

TABLE A2 Common pip3 package

numpy 1.18.1	tensorboard 2.1.0
Pillow 7.0.0	tensorflow-estimator 2.1.0
opencv-python 4.2.0.32	termcolor 1.1.0
Keras 2.3.1	tf-slim 1.0
Keras-Applications 1.0.8	tf2cv 0.0.10
Keras-Preprocessing 1.1.0	

TABLE A3 The pip3 packages used for the framework installation

Framework	CPU	GPU
pyTorch	torch 1.4.0+cpu	torch 1.4.0
pyTorch	torchvision 0.5.0+cpu	torchvision 0.5.0
Paddle	paddlepaddle 1.6.3.post107	paddlepaddle-gpu 1.6.3.post107
MXNet	mxnet 1.6.0	mxnet-cu101 1.6.0.post0
TensorFlow	tensorflow 2.1.0	tensorflow-gpu 2.1.0

The operating system is an Ubuntu 18.04.4 LTS with the specific software of Table A1 and the Python 3 packages of Table A2. The virtual python environment of each framework is reported in Table A3 for the CPU and the GPU.