Benchmark of TensorFlow 2.1 and MXNet 1.5.1*

From Fundamental Operations to End-to-End Training

Matteo Paltenghi
Database Systems and Information Management (DIMA)
Technische Universität Berlin
Berlin, Germany
matteo.paltenghi@campus.tu-berlin.de

Kaoutar Chennaf
Database Systems and Information Management (DIMA)
Technische Universität Berlin
Berlin, Germany
chennaf@campus.tu-berlin.de

Behrouz Derakhshan
DFKI
Technische Universität Berlin
Berlin, Germany
behrouz.derakhshan@dfki.de

ABSTRACT

Thanks to the latest advances in deep learning (DL), neural networks have become a crucial component to obtain state-of-the-art performance in a large variety of artificial intelligent tasks. Parallelly to the proposal of novel architectures to solve problems more efficiently, new deep learning libraries have evolved through time to adapt to the new features required by the research and industry communities. One of the very first libraries in this context is the well-known TensorFlow by Google, beside this one, other new DL frameworks were born in the following years: MXNet, PyTorch, Caffe, etc. The purpose of this benchmark is to compare the performance of two of these libraries that are showing a steady growth in their APIs and community: TensorFlow and MXNet. Our benchmark is one of the first to study TensorFlow 2.1. Moreover, it is not only focused on benchmarking a complete network but also low-level API through common fundamental operations in neural networks. We also explored the possible optimizations available in both frameworks with an evaluation of them. Besides the popularity of TensorFlow, we were able to prove that MXNet deserves a lot more attention considering its very competitive performance and speed.

KEYWORDS

Benchmark, Deep learning Frameworks, TensorFlow 2.1, MXNet, Fundamental Operation, Operator, GPU, CPU

1 Introduction

The deep learning (DL) research has progressed from the first perceptron to the modern convolutional neural networks. Neural networks (NN) have allowed us to get some unique insights into whichever task they were designed for. NNs were inspired by the way nervous systems process information. They have proved a remarkable ability to extract complex patterns and trends and derive meaning from unstructured data. They have been employed in various fields ranging from computing, to medicine and economics. This section aims to introduce the basics of neural networks, describe existing frameworks, give the history of TensorFlow and MXNet, state our contribution, and finally outline the report structure in the following sections.

*Our repository is accessible through the following link: http://github.com/MattePalte/Tensorflow-VS-MXNet-Benchmark

1.1 Neural Networks and Deep Learning

Neural Networks, also referred to as Artificial Neural Networks (ANN), represent an information processing paradigm whose key element is its structure based on a nervous cell, also called a perceptron. Hence, a neural network is composed of a set of artificial neurons interconnected between each other through layers, with each neuron having a weight and an activation function [6]. These weights determine the impact of one neuron on another one. When at least three layers are composing an ANN, we refer to a Multi-Layer Perceptron (MLP), which is consisting of an input and an output layers, and at least one hidden one. The input data traverse these layers and are transformed by weights and activation functions until they reach the final output layer. From one single perceptron that was first designed in 1958 to today's MLPs, ANNs architectures are based now on these three following characteristics: the number of layers in the network in addition to the number of neurons in each of them, the learning mechanism that is used to update the weights of neuron, and finally the activation functions that are used in different layers. Deep Learning refers to concepts and techniques related to neural networks, with the characterization deep referring to networks with a large quantity of hidden layers and parameters. The first successful neural network dates back LeNet [9], which is a convolution neural network meant to classify images and it is based on convolution, which is mainly a mathematical operation that allows to extract features from images and can be reconduct to a matrix multiplication.

1.2 Deep Learning Frameworks

There are currently more than ten DL frameworks, with a market share mainly being divided between Caffe, Pytorch, TensorFlow, and MXNet. Each of these frameworks has its strengths and weaknesses. Caffe is a widely used library which ported Matlab implementation to C and C++ of convolutional NNs. It is hence intended for computer vision and not for text, sound, or time series data [17]. Like many other frameworks, Python is the language chosen for its API. Its strengths are basically training models without writing any low-level code by fine-tuning existing networks. However, it is not extensible and does not offer commercial support. PyTorch is a Python version of Torch which was open sourced in January 2017 by Facebook. Since its introduction, it quickly became the favourite DL framework of machine-learning researchers, since it allows to easily build certain complex architectures. It offers dynamic computation graphs, which enable the processing of variable-length input and output. PyTorch's main strengths are its modular pieces that are easy to combine, in addition to enabling developers to customize their own layers and run them on GPU [17]. However, it also lacks commercial support and the documentation remains spotty. MXNet is an Apache open-source machine-learning framework. It has APIs in Python, R, and Julia among other languages and allows fast, flexible, and scalable model training. It is supported by public cloud providers such as Microsoft Azure and Amazon Web Services (AWS). Its main contributors are Amazon, Wolfram Research, MIT, Microsoft and other prestigious academic institutions. TensorFlow is a free open-source math library developed initially for internal use by the Google brain and was released under the Apache License 2.0. It is a replacement of Theano and currently supports differentiable programming and dataflow. It has a flexible architecture which allows easy deployment of computations across numerous hardware such as GPUs, CPUs, and TPUs. Nowadays, TensorFlow remains the framework of choice of most research institutions and multinational companies across the globe such as eBay, LinkedIn, and SAP.

1.3 Our Contributions

Through this work, we studied TensorFlow 2.1 and MXNet 1.5. Our main metric was the time it takes to train a network -LeNet in our case- as well as to execute fundamental operations in order to have a clear idea on potential bottlenecks that each framework might have. We further explored optimization techniques in each framework along with the benefits of using GPU over CPU setup.

1.4 Report Organization

This benchmark report is organized as follows: section 2 addresses related work, section 3 addresses the main high-level differences between the frameworks, section 4 describes the methodology our benchmark as well as hardware and software setup, in addition to presenting the results of each benchmark. In section 5 we discuss our results and address some limitations and future work, and finally in the conclusion we summarize our main take-aways.

2 Related Work

With the undeniable accuracy of neural networks, benchmarking deep learning frameworks have gained a wide interest with the development of new frameworks and the need to always achieve the highest performance. Numerous benchmarks attempted to evaluate different aspects of DL frameworks, such CPU versus GPU hardware setups, parallelized versus single node, training time, inference time, and very few of them analysed end-to-end neural networks. DL benchmarking approaches can be mainly categorized in three categories: (i) operator level, where single low-level operators are isolated and profiled; (ii) mini-batch level, where a single mini-batch is analysed during either training or inference; and (iii) workload level, which represent a holistic end-to-end evaluation of entire workloads of training or inference [1].

Rysia[1] is one of the most recent benchmarks which aimed to show that GPU-accelerated executions widely depend on the model architecture, the workload, and the mini-batch size, with the latter yielding the highest performance gain when it's large. MXNet, Pytorch, and TensorFlow were the main frameworks of interest in this benchmark, with the analysis of three complete neural networks. According to [1], no platform outperforms any other for every tested workload. Moreover, "symbolic setups with static computational graphs do not guarantee better performance than imperative setups with dynamic graphs" [1]. For CPU-restricted hardware setups, the performance gain depends largely on the workload model as well as the platform of choice, instead of depending on the degree of parallelism involving a higher number of CPU cores. Among the benchmarked framework, TensorFlow recorded the highest possible scalability for such CPU-restricted hardware setup. Another benchmark worth mentioning is Fathom [2] which originated from the assumption that deep learning platforms performance is mainly dominated by the fundamental operations -such as matrix multiplication or convolution- that make up the computation bulk for any given workload. This assumption was the reason why we also tested fundamental operations in our own benchmark. The authors were indeed able to verify it by providing statistics for certain workloads - which show the time spent per operator for every one of these workloads [2]. Surprisingly, performance overhead created by these operators varies across workloads. Operators that have highest computation benefit from higher degrees of parallelism, while other operators - with short computation- will worsen because they will spawn new threads creating useless overhead for the distribution.

Deepbench [3] was a benchmark that attempted to find which hardware provides the best performance for fundamental operations in neural networks. The authors implemented by hand a number of low-level operators in C++ with commonly utilized libraries in deep learning such as NVIDIA Cuda and Intel MKL. Matrix multiplication, recurrent network operators, and convolution are the main analysed operators. The goal of this benchmark is to raise -amongst software developers and hardware vendors- awareness about bottlenecks in training and inference workloads.

Microsoft conducted a benchmark as well in [4], where twelve deep learning platforms including TensorFlow and MXNet were covered. All experiments were executed in GPU accelerated hardware within Microsoft Azure cloud environment, and mainly covered feature extraction and image recognition in convolutional networks, and sentiment analysis in recurrent networks. MXNet recorded the fastest training runtime for both workloads.

S.Shi et al. benchmark [5] compared a variety of deep learning frameworks on both CPU-restricted and GPU-accelerated setups on the mini-batch level with an exclusive focus on training. TensorFlow and MXNet were among the platforms that were analysed. With the time duration of one processing iteration of a mini-batch input was the performance evaluation metric, the authors analysed various workloads of convolutional, recurrent, and feed forward architectures. The benchmark shows -like other previously described ones- that performance speedup is limited with an increasing number of CPU cores, for all platforms and all analysed workloads. TensorFlow recorded again the highest possible scalability for such hardware setup. On single GPU setups, MXNet performed best for convolutional architectures with many-layered networks, while CNTK outperformed all other platforms for recurrent networks. For feed forward networks, Cafe, CNTK, and Torch outperformed MXNet and TensorFlow. MXNet again scaled best in multi-GPU setups for convolutional architectures, while CNTK scaled best for feed forward architectures. A significantly higher throughput rate and convergence speed were recorded across all platforms and workloads for data-parallel multi-GPU setups [5].

3 High Level Comparison

This section aims at introducing some necessary background information about the two frameworks. It addresses their history, mains features, pros & cons, architecture and API, the programming paradigms of each framework, and the optimizations available in each one of them.

3.1 History

TensorFlow is a Google product that was initially released in November 2015. A little bit before this time, in March 2015, Keras -a neural network library was released by its creator François Chollet. By the end of the same year, Amazon and Microsoft released the first version of MXNet. Two years later, in 2017, Keras was integrated for the first time in TensorFlow 1.1 in April, and In October, Gluon was implemented within MXNet. The last more significant changes of both frameworks happened in the last months, with MXNet 1.5 in June '19 and TensorFlow 2.1 in January '20.

3.2 Main Features and Pros & Cons

With both being first released in the same year, MXNet and TensorFlow still have some major differences which in the end converged to the same programming paradigm. Both frameworks were written in C++. While TensorFlow only supports Python, MXNet provides rich support for many languages such as R, Scala, Scala, and C++ in addition to Python. TensorFlow JS allows the deployment of models in JavaScript environments in both frontend and Node.js backend, in addition to defining models in JavaScript and training them directly in the browser through a Keras-like API. TensorFlow offers monitoring and visualization as main components of its framework through the Tensorboard interface. Both frameworks support CPU, GPU, and mobile hardware, with TensorFlow supporting TPUs as well when running on the Google Cloud Service. For mobiles devices, TensorFlow Lite enables on-device inference with low latency. Both frameworks offer models serving for the deployment of the trained models, as well as distributed execution, and support declarative as well as imperative programming. Moreover, they both aim to offer high-performance and scalable model training and deployment. TensorFlow struggles with poor results for speed in many benchmarks when compared to PyTorch or CNTK [1,3,5] while MXNet suffers from a very small community and remains unpopular among in the research field. Even if MXNet is flexible, fast, and efficient in terms of running DL algorithms, TensorFlow remains the dominating DL framework by offering extensive documentation and guidelines and having a wide community supporting it.

3.3 Architecture

The architectures of the two frameworks operate on top of the same hardware (i.e. CPUs, GPUs, Android, iOS) with the exception of TPUs that are an exclusive Google ASIC (application-specific integrated circuit) dedicated to Deep Learning. The other common factor is the presence of multiple language bindings, with MXNet that rules the comparison by having a larger variety of supported languages frontends. In terms of core, both of them have a C++ nucleus. Worth to be noticed as a separate component for TensorFlow is XLA (Accelerated Linear Algebra) that will be explained in the optimization section. On top of the TF core, the main language binding is the Python frontend, that is the de-facto standard for every deep learning library. On top of it TensorFlow has the following API: Dataset API to build input pipelines, Layers API and Keras API to construct the network and Estimator API to abstract the main operation of training, evaluation and prediction. On the other hand, MXNet has its counterparts: DataLoading API to iterate over your data, KVStore API to achieve distributed communication, Symbolic API and Executor API to create computation graph and execute it respectively, NDArray API to mimic general linear algebra operations and tensor manipulations, and Gluon API family to build networks in different fields (Computer Vision, Time Series, Natural Language Processing).

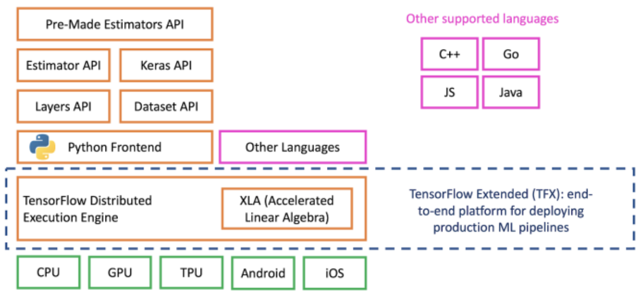

Pre-Made Estimators API
Estimator API
Keras API
Layers API
Dataset API
Python Frontend
Other Languages
Other supported languages
C++
Go
JS
Java
TensorFlow Distributed Execution Engine
XLA (Accelerated Linear Algebra)
TensorFlow Extended (TFX): end-to-end platform for deploying production ML pipelines
CPU
GPU
TPU
Android
iOS

Figure 1: TensorFlow Architecture [7]

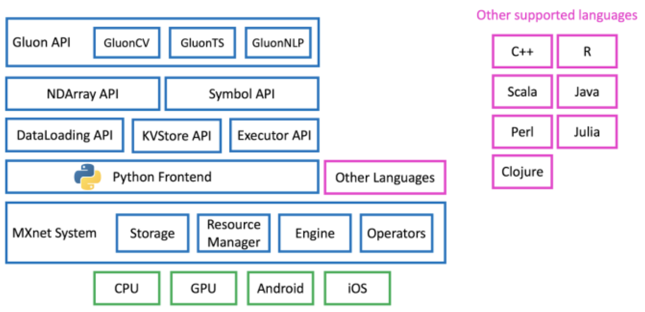


Figure 2: MXNet Architecture [8]

3.4 Programming Paradigms

Before diving deeper into the APIs of each framework, some prior definitions of programming paradigms are needed to give a better understanding. On one hand, symbolic programming -also referred to as declarative- refers to a code containing three main parts: (i) the definition of the computational graph -in which all the operations of the neural network are explicitly declared and chained one to another, (ii) the compilation of the computational graph into an executable program, and finally (iii) the execution of the compiled program when fed with the required input. Imperative programming, on the other hand, is very similar to the natural way a Python program is expressed. Every command is executed one after the other and each computation is performed at that moment without any delay. It is usually easier to debug as no compilation is needed. Each of these paradigms has its pros and cons. For imperative programming, one of its main advantages is the fact that it is flexible and versatile when trying new neural network architectures or configurations. Furthermore, the possibility of accessing all relevant intermediate variable values makes it easier to debug. While for symbolic programming, since it should be compiled, it enables some optimizations and further checks as a main advantage. However, it remains less intuitive and less easy to debug.

Historically, in the context of Deep Learning frameworks, symbolic programming was the first one adopted by TensorFlow. Then, PyTorch -which adopts imperative programming style- started gaining popularity in particular in research settings due to its flexibility. In this context, MXNet which is relatively a newcomer, embraced mainly the imperative style, but with a great focus since the beginning on efficiency. Nevertheless, both frameworks understood that there is no definitive winner between symbolic and imperative style. Therefore, they started to get the best from each one of them: MXNet with the so-called Hybrid programming style, explained in the optimization section, and TensorFlow with the transition in its latest release TF2 to eager programming by default, allowing them to reach a level of intuitiveness comparable with the other competitors.

3.5 Available Optimizations

Our analysis of the two framework APIs led us to the exploration of methods to increase their performance. Therefore, we deeper investigated various optimizations available in the two frameworks. The two most important ones were:

- XLA (Accelerated Linear Algebra) which is a domain-specific compiler for linear algebra that can accelerate TensorFlow models with potentially no source code changes [12].
- Hybridize is a way to enable the MXNet programmer to use both imperative and symbolic style, imperative while programming and once he is ready to train the model switch to symbolic programming to exploit the efficiency of the computational graph [13].

The XLA optimization process is modular and composed of two stages: an optimization that is target independent and a second one that depends on the specific platform of choice. Some of the optimizations that are performed are: common subexpression elimination (CSE), target-independent operation fusion, buffer analysis for allocating runtime memory for computation.

While XLA can be activated for every network, on the other hand Hybridize needs to have a network object composed of HybridBlock, otherwise it will not be possible to convert it in an optimized computational graph. Therefore, we investigated the limitation of the HybridBlock with respect to normal Blocks. A Block is the base class for all neural network layers and models and in the Hybrid version the following operations are forbidden: (i) access a specific element in a tensor, (ii) usage of conditional logic on data type, (iii) direct assignment of an element in a tensor, (iv) access of context (i.e. CPU or GPU), (v) access the shape of a tensor. To activate XLA you need to add the following line after your import section tf.config.optimizer.set_jit(True) and to make it runs on CPUs you need to use the additional flag before running your python script: TF_XLA_FLAGS=”--tf_xla_auto_jit=2 --tf_xla_cpu_global_jit”. Conversely, for MXNet, once your model is Hybrid (e.g. HybridSequential) you just need to call the hybridize method on your model object.

4 Benchmark methodology

The focus of the whole work conducted through this benchmark was put on the entire time required for training (local or cloud), since it is usually the most relevant metric for data scientists. As a matter of fact, once the architecture is given the user wants to have a working model in the least possible time. This section gives an overview of the benchmark, describes its hardware and software setup, and gives the details of each conducted benchmark and its results.

4.1 Overview

Our review of the presented end-to-end benchmarks evaluating the training process of a neural network showed us that they commonly measure the required time to reach a specific accuracy with a given model architecture and dataset. We believe that an exclusive consideration of a specific accuracy as a finishing line does not guarantee comparability between platforms. In fact, depending on various factors -such as hardware specifics or optimization parameters, one platform might need a higher or smaller number of iterations to reach that certain level of accuracy. Rysia [1] uses low-level APIs for platform specific implementations to ensure comparability between frameworks. Such approach is hardly possible now with TensorFlow 2.0 as they are mostly encouraging Keras as their main High-level API. We hence propose functional equivalence through aligned high-level layers in both frameworks, and with a predefined number of epochs and mini-batch size. We then evaluate the time it takes for a framework to finish the number of iterations, along with the level of accuracy reached.

We structure our benchmark in four different sections:

- Benchmark A: Benchmarking of low-level fundamental operations on both GPU and CPU
- Benchmark B: End-to-End benchmark of LeNet Convolutional Neural Network training on CPU and GPU

- Benchmark C: End-to-end benchmark of XLA and Hybridize on LeNet on CPU and GPU
- Benchmark D: Benchmark of TensorFlow optimizations: eager execution and XLA.

4.2 Hardware and Software Setup

We benchmarked TensorFlow 2.1 and MXNet 1.5 on both hardware. We used the latest versions of Python (3.5.2), Pandas (0.23.3), Numpy (1.18.1), Sklearn (0.22.1). For all the GPU setup, we used a machine provided by TU Berlin, offering 62GB of RAM, AMD Opteron(tm) Processor 6376 with 32 cores, and NVIDIA GTX 980 with a CUDA version of 10.2. For CPU experiments in Benchmark B and D we used a virtual machine with core i5 and 4GB of RAM while for the ones in Benchmark A and D we used the TU Berlin configuration with the AMD processor.

4.3 Benchmark A: Low Level Operations

Before starting to analyse a full-fledged neural network, we spent considerable time exploring the fundamental operations that are the basis for all neural network operations. We believe that if a framework does well on fundamental operations, it will probably achieve the same, if not better, in a complete network. The selected pool of operations was tested on both libraries with CPU and GPU runtime. The operations benchmarked are the following:

1. Vector creation
    a. starting from Numpy arrays
    b. starting from Python lists
2. Activation functions
    a. Sigmoid
    b. ReLu
    c. Hyperbolic Tangent (tanh)
    d. SoftReLu
    e. SoftSign
3. Dot Product
4. General Matrix Multiplication (GEMM)
5. Flattening of a vector
6. Matrix normalization

The absence of the convolutional operation is due to the lack of a specific function for it in the low-level API, to the best of our knowledge. Nevertheless, we managed to find that the convolutional operation is usually implemented with the im2col algorithm that is used to transform it into a general matrix multiplication, which is usually increasing the memory footprint due to the conversion to a different data structure called Toeplitz matrix [9]. Also, in cuBLAS -the optimized linear algebra library used by NVIDIA GPUs- the implementation of convolution is obtained via GEMM, a general matrix multiplication [10].

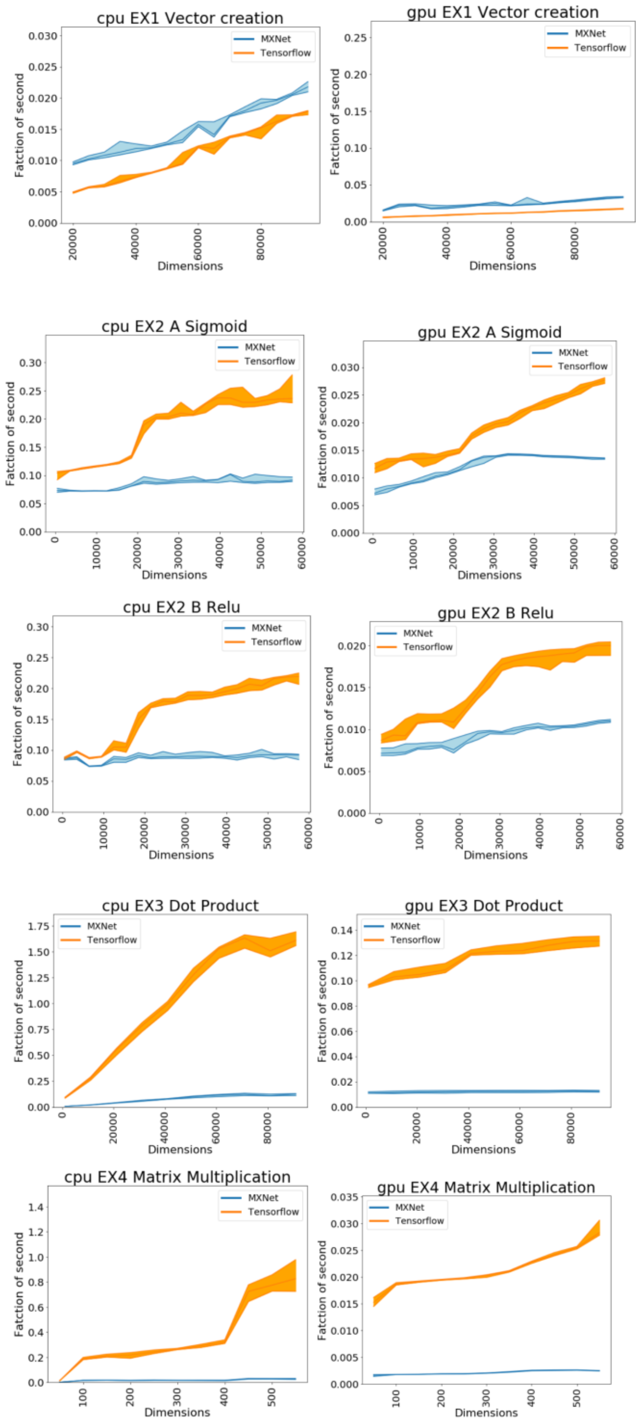


Figure 3: Low Level Operations Benchmark Results

From these diagrams we can notice that MXNet, apart from vector creation, is way faster than TensorFlow on both CPUs and GPUs, especially in higher dimension. For CPU there is mainly a difference in the scalability of the two, with TensorFlow that degrades faster; while for GPU they both have a lower slope but they are on two different runtime level with MXNet leading the duel.

4.4 Benchmark B: End-to-end Training of LeNet

Part of the great resonance gained by Deep Learning is the improvements achieved in the computer vision field, therefore we choose to benchmark the training of one of the first successful Convolutional Neural Networks called LeNet. This architecture was able to achieve state of the art performance in character recognition. Based on its original paper [11] by Yann Lecun, we used the MNIST dataset composed of 60000 images of size 28*28 whose training was done on 48000 of them, as the rest was kept for validation. The implemented network is composed of the following layers: C(6,5,1) - P(2,2) - C(16,3,1) - P(2,2) - Flatten - FC-FC-FC. C(6,5,1) refers to a Convolutional layer, with a feature map of size 6, a kernel size of 5*5, and a stride of 1. P(2,2) refers to an Average Pooling layer with a kernel size of 2 and a stride of 2. Finally, FC refers to a fully connected layer. The activation function used in each convolution layer and the two first fully connected ones is ReLu. The main focus of this first benchmark is to evaluate the training performance -in terms of time and accuracy- of the two frameworks in the two runtimes, CPU vs GPU. We guaranteed functional equivalence by defining the same layers with the same hyperparameters for each one for both frameworks. We used Adam optimizer, along with a learning rate of 0.001, and Softmax-Crossentropy as a loss function. We set the mini-batch size to a constant of 200, while we varied the number of epochs to three values: 5, 10, and 15. For each number of epochs, we ran the experiment 5 times, and the values shown below were the obtained average values from the 5 experiments of each number of epochs.

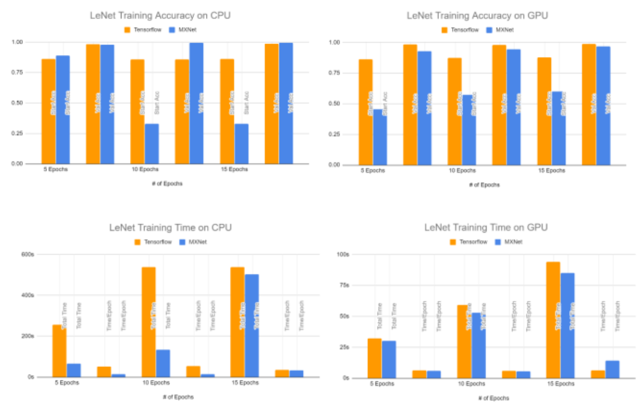


Figure 4: LeNet End-to-End Training Time and Accuracy

Concerning the bar chart of accuracy, each bars on the left for each number of epochs refer to the accuracy obtained in the first epoch, while the bars on the right refer to the validation one. We can clearly notice that MXNet has a slow convergence to the validation accuracy while TensorFlow starts from close values and doesn't need to increment it much. In the end of each training, they both have similar accuracy rates, with TensorFlow slightly exceeding MXNet. Concerning the training time, each two bars on the left of each epoch refer to the total time while the ones on the right refer to time/epoch in seconds. For this part, we can clearly notice the difference in speed between the two frameworks, which becomes smaller on GPU. When increasing the number of epochs, TensorFlow seems to catch up and get close to the training speed of MXNet. The benchmarks [5,2] confirm that MXNet is one the best frameworks for CNNs, which could explain the results we obtained from this benchmark. However, further explanations were not provided as of why it is the case.

4.5 Benchmark C: MXNet and TensorFlow Optimizations Analysis

In this benchmark we wanted to evaluate the effect of the two optimization -XLA for TF and Hybridize for MXNet- on the LeNet architecture both in the CPU and GPU setting. In total 8 configurations were tested and summarized in the following scatterplots (true means that optimization is switched on, false the opposite). The workload tested had the following common characteristics to each configuration:

- LeNet architecture described in Benchmark B.
- MNIST dataset (48'000 train, 10'000 test).
- Running for 15 epochs, with a mini-batch size of 200.
- Record of Runtime for the training and record for the final accuracy on test set.

For each configuration we collected at least 5 data points.

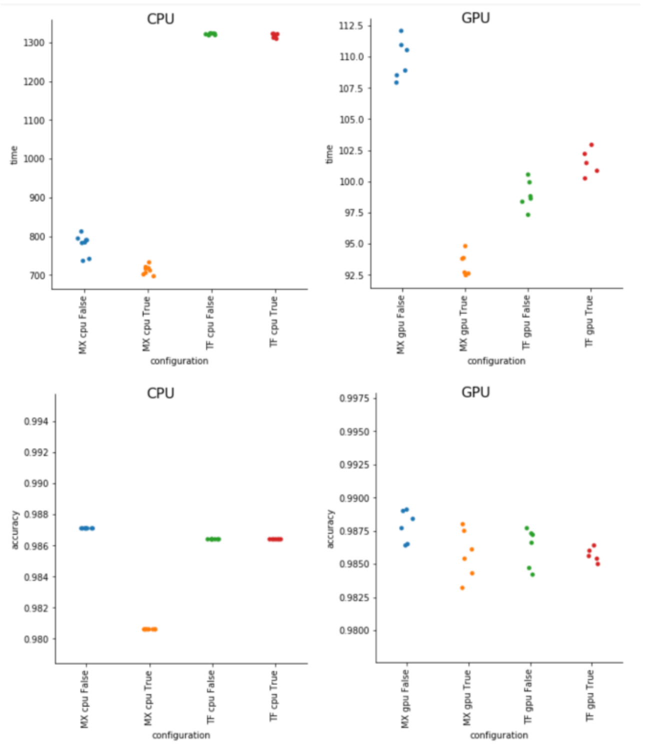
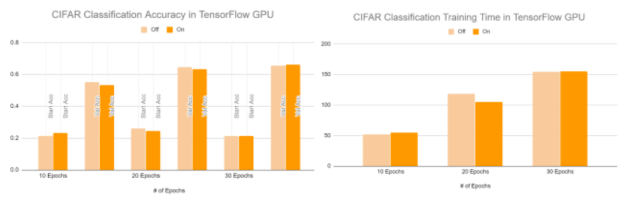


Figure 5: TensorFlow and MXNet Optimizations Benchmark

By analysing the resulting scatterplots, we can see that MXNet is significantly faster on CPU (speed up of ca. 1.73x) while on GPU it is faster only with the optimization switched on. In terms of accuracy, in the CPU setting we can notice that all the experiments are fully reproducible leading to the same accuracy in every run, while for GPU this is not the case due to the underlying hardware approximation probably. In terms of overall accuracy, we notice that MXNet without optimization (MX cpu False) is always getting the highest score. Another important fact derived from the runtime scatterplot of GPU is that XLA is not speeding up, but on the contrary it is slowing down the train. One possible interpretation can be that the framework is trying to optimize the network but perhaps the network structure cannot be optimized and therefore there is the overhead for this first check. For MXNet it can be noticed that the optimization is leading a different distribution of final overall accuracy, whereas TensorFlow is able to maintain a better operational equivalence between the normal setting and the XLA enabled. In this regard it can be said that MXNet with its optimization is trading off runtime with accuracy.

4.6 Benchmark D: TensorFlow optimizations

We wanted to explore in more detail why TensorFlow is slow in training. We found that the issue may not be about being slow in general, but more about the second version being slower than the first one, which is the one that made it gain most of the popularity it has now. So, we dug deeper into the implemented optimizations within the framework. We hence re-evaluated XLA along with eager execution that is offered by default in TensorFlow 2.0 and above. This time, we did not compare the performance with MXNet, but we compared each optimization when it's on to when it's off, in order to measure the performance gain for each one of them. Eager execution refers to an imperative programming environment which evaluates operations immediately. Operations hence "return concrete values instead of constructing a computational graph to run later" [15]. This approach makes debugging easier. The eager execution benchmark was conducted on the LeNet architecture described in Benchmark B, with the same methodology, meaning that for each number of epochs, 5 experiments were run, and the average value is the one considered for both time and accuracy. While the XLA one was conducted on a defined network by TensorFlow [16] to classify the CIFAR-10 dataset composed of 60000 images of size 32*32 and 10 classes. The authors claim that "On a machine with a Titan V GPU and an Intel Xeon E5-2690 CPU the speed up is ~1.17x" [16], which remains considerably small. We wanted to verify this assumption through our experiments. The dark orange bars refer to when the optimization is On, while the bright ones refer to when it is Off.



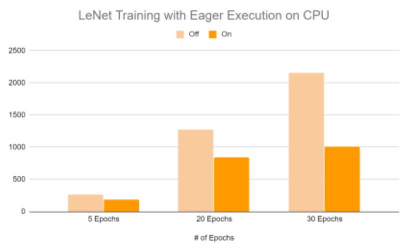


Figure 6: TensorFlow Optimizations Results

Concerning XLA, the closest value we got to 1.17 of speedup from our experiments was 1.15 for 25 epochs. In fact, the first epoch was always the one that would take an average of 2.5 times the average time for an epoch to finish. For the rest, XLA doesn't seem to reduce much the training time and accuracy remains unchanged as well. Eager execution on another hand, seemed to be doing a pretty good job, by reducing considerably the amount of time required for the training without affecting accuracy.

5 Discussion

5.1 Results Summary

From the results it appears that MXNet is most of the time faster than TensorFlow both on CPU and GPU. We investigated further the reason of this behaviour and some motivation can be linked to the low-level implementation of the fundamental operations, since also that benchmark shows an advantage of MXNet. Nevertheless, CPU-based distributed environments are not efficient in terms of speed for most frameworks when compared to GPU ones [1]. As a matter of fact, we found a confirmation in an official publication [14] that, in contrast to TensorFlow, MXNet is calling tuned BLAS/LAPACK libraries directly. We appeal to this official source as the best possible explanation since the alternative way of manually inspecting the code was time consuming and could even lead to wrong conclusions due to an ease of misinterpreting someone else's code. Concerning accuracy, both frameworks were able to achieve high levels of accuracy with slight differences. Benchmark B allowed us to distinguish TensorFlow as being better for accuracy while MXNet was better for speed.

5.2 Reproducibility

One of the first questions that arises once you look at the accuracy result is why they are not the same since both architectures and operations should be the same on both frameworks. As a matter of fact, the two frameworks are handling some approximation under the hood, like the process of auto differentiation, that makes them achieve relatively different results even when the configurations and optimizers are exactly the same. Beside this work, we started adapting Rysia benchmarking framework to the new TF2 but even after that we realized that the perfect match of accuracy between the two framework is impractical.

5.4 Limitations and Lessons Learned

On one hand, throughout this work, we have explored many other options in order to explain the difference in performance between the two frameworks. The first one was Psutil, which is a function supposed to return the resource consumption whenever it is called. However, it remained unhelpful in our case since it needed to be called during each layer of the training in order to get meaningful results. cProfile on another hand, allows to find out which fundamental operation is being called during training. However, TensorFlow core would obfuscate the details since we would only know that a core operation is called but not which one. We also tried LeNet on TPU in the Colab environment, however we got a worse performance than on GPUs, probably because it is still experimental in the Colab environment. On another hand, we learned that both frameworks have an equivalent and intuitive network definition, with TensorFlow having an easier fit API. We also learned that the shape of tensors is the cause of most of the problems in a network. Last but not least, we learned some valuable teamwork skills while applying the ones we already had, which was another main takeaway from this work.

5.5 Future Work

There are numerous other options that we could have explored in our benchmark if we had more time. The first ones are tf.data.Dataset along with ImageDataGenerator in TensorFlow to see how much performance gain they offer. Another option would be to explore TPUs on Google Cloud and measure their speed compared to GPU settings. XLA also seemed like a promising optimization, despite the results we got, so we wanted to further investigate it on more complex networks. Also, exploring other network architectures on both frameworks such as LSTM would bring more results to analyse or support our claims. Furthermore, extending this benchmark to other DL frameworks such as Pytorch on both fundamental operations and end-to-end training would give a global view on the stand of these frameworks. Another promising approach for future work, would be to identify factors that cause divergence in speed and more importantly accuracy on both frameworks.

5.6 Work Division

In our team, Matteo took care of the design and execution of the benchmarking of fundamental operations and the harmonization of the two implementations of LeNet in the two frameworks in order to benchmark the available optimization techniques. The code of the experiments is available on Github and fully reproducible up to the chart creation to display the results. He also investigated the optimization techniques in general with a particular focus on Hybridize of MXNet and took care of promptly debugging most of the problems related to Github and GPU usage that our team has faced. Kaoutar, on the other hand, took care of the end-to-end benchmark of Lenet. She also explored available optimization techniques, especially on TensorFlow, in order to explain its slow performance. In addition, she reviewed other similar benchmarks and was able to compare our results with them to clearly confirm our obtained results. As for the rest, meaning the report and the presentations, the work was divided equally in a spontaneous way. Clearly, this is a winning teamwork.

6 Conclusion

Throughout this benchmark, we were able to explore the latest versions of TensorFlow and MXNet both on CPU and GPU. In our end-to-end benchmarks, our main metrics were the time for the network to train for a certain number of epochs, in addition to the accuracy reached. We also explored fundamental low-level operations in both frameworks and on both hardware, for which only the time was taken into account. Furthermore, we explored optimizations available on both frameworks, mainly Hybridize for MXNet, and XLA along with eager execution for TensorFlow. We could clearly see a pattern of MXNet being generally faster than TensorFlow. Besides, we were able to guarantee functional equivalence by constructing the same network architecture with the same hyperparameters on both frameworks. Our literature review allowed us to compare our results, and we can clearly state that TensorFlow, despite its popularity, isn't the fastest DL framework. Moreover, if one is looking for speedup, GPU settings remain the best compared with distributed CPU settings [3]. We hope this work to be useful for future users of these frameworks.

ACKNOWLEDGMENTS

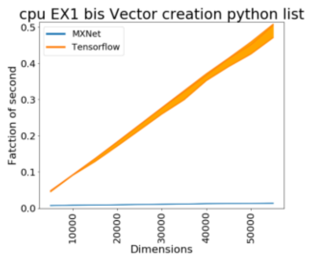
Thanks to Behrouz for his valuable very insight, in addition to his attention, availability, and flexibility. Thanks also to Clemens for the computing resource provisioning and to Sergey for the useful advice in terms of directions to take.

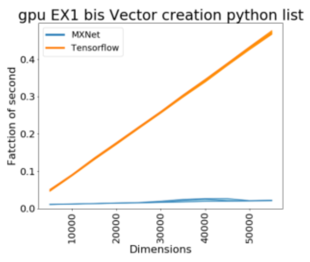
REFERENCES

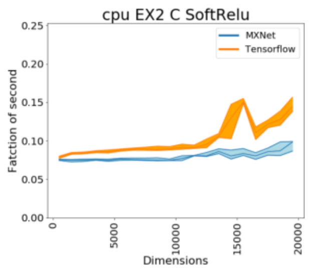
1. Rysia Benchmark. URL: https://github.com/vdeuschle/rysia. Accessed: 2019-11-11
2. Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In Workload Characterization (IISWC), 2016 IEEE International Symposium on, pages 1-10.IEEE, 2016.
3. DeepBench. URL https://github.com/baidu-research/DeepBench. Accessed: 2020-01-02.
4. Comparing Deep Learning Frameworks: A Rosetta Stone Approach. URL: https://blogs.technet.microsoft.com/machinelearning/2018/03/14/comparing-deep-learning-frameworks-a-rosetta-stone-approach/. Accessed: 2020-01-02.
5. Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In Cloud Computing and Big Data (CCBD), 2016 7th International Conference on, pages 99{104. IEEE, 2016.
6. Mabrouka AL-Shebany et al. Int. Journal of Engineering Research and Applications www.ijera.com ISSN : 2248-9622, Vol. 4, Issue 2(Version 1), February 2014, pp.07-12
7. Abadi, Martín, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).
8. Chen, Tianqi, et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems." arXiv preprint arXiv:1512.01274 (2015).
9. LeCun, Yann, et al. "Learning algorithms for classification: A comparison on handwritten digit recognition." Neural networks: the statistical mechanics perspective 261 (1995): 276.
10. (source:https://arxiv.org/abs/1907.02129 and https://docs.nvidia.com/cuda/cublas/index.html)
11. LeCun, Yann, et al. "Learning algorithms for classification: A comparison on handwritten digit recognition." Neural networks: the statistical mechanics perspective 261 (1995): 276.
12. XLA. URL https://www.TensorFlow.org/xla. Accessed: 2020-02-02.
13. Hybridize. URL https://beta.MXNet.io/guide/packages/gluon/hybridize.html. Accessed: 2020-03-03.

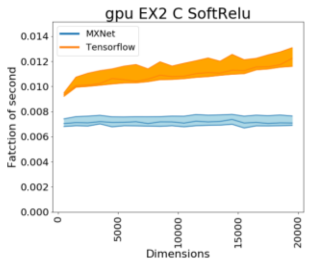
14. Seeger, M., Hetzel, A., Dai, Z., Meissner, E., & Lawrence, N. D. (2017). Auto-differentiating linear algebra. arXiv preprint arXiv:1710.08717.
15. Eager Execution. URL https://www.TensorFlow.org/guide/eager . Accessed: 2020-03-05.
16. Classifying CIFAR. URL https://www.TensorFlow.org/xla/tutorials/autoclustering_xla . Accessed: 2020-03-05.
17. Comparison of AI Frameworks. URL https://pathmind.com/wiki/comparison-frameworks-dl4j-tensorflow-pytorch. Accessed: 2020-01-14.

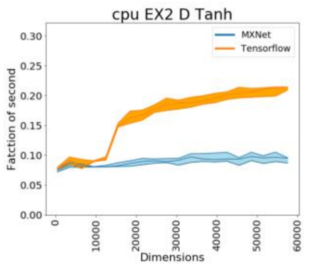
APPENDIX

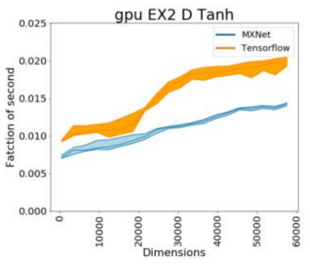


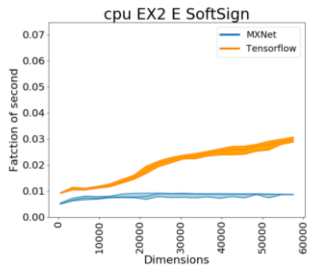


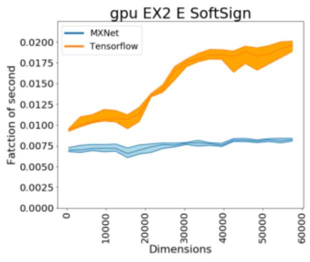


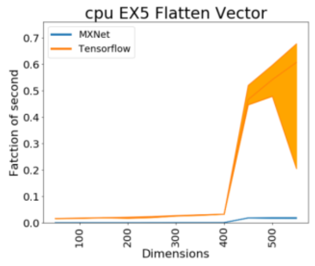


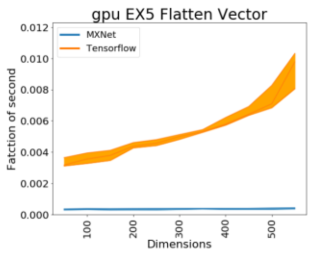


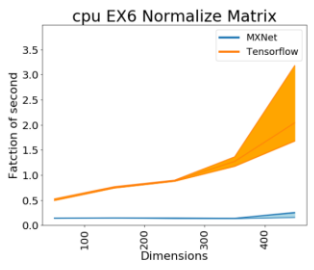












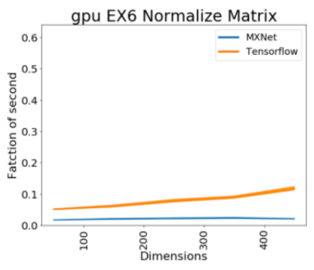


Figure 7: Low Level Operations Benchmark Results