UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Improving the performance of a deep learning framework on high-performance computing (HPC) systems

Document:
Report

Author:
Martí Llopart Font

Director /Co-director:
Bernardo Morcego / Ramon Pérez

Degree:
MÀSTER UNIVERSITARI EN GESTIÓ D'EMPRESES DE TECNOLOGIA I D'ENGINYERIA

Examination session:
Autumn/Winter, 2022-2023

MASTER FINAL THESIS

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Abstract

In this work we intend to improve the performance of the library Pytorch 1.13.1 in High-Performance Computing (HPC) applications for Central Processing Units (CPUs). The Pytorch framework is an open-source library that is intended to ease the burden of programming neural networks for Machine Learning (ML) purposes. Since its creation by Facebook, Pytorch has offered parallelism features. However, these options are fixed, meaning that they can't be changed during the training and inference processes of neural networks. In these processes, some sections of a network have more tasks that can be parallelized than others. Therefore, if the parallelism parameters offered by Pytorch can only be selected in a fixed way, the program won't run at its best efficiency for the most part. A solution to that problem would be to dynamically change the parallelism features in Pytorch, according to the nature of the neural network architecture at a certain layer. To showcase this, we selected a type of neural network called Long Short-Term Memory (LSTM), which has varying widths. In other words, we chose a neural network containing sections where many operations can run in parallel, and sections where only a few operations can run in parallel, or none. The network we've used is currently the state-of-the-art for NLP (Natural Language Processing), making it the perfect network to study for HPC applications. Such applications contain tasks like machine translation, text generation and next-word prediction. In the present study, we've developed a use case where an LSTM network is used in an inference process. In the use case, the network ran using three different settings: without any parallelism configurations, with fixed parallelism configurations and by dynamically tuning the parallelism configurations. The objective of this work is to show that by approaching parallelism in a dynamic way, many Pytorch applications can see a huge improvement in terms of performance. Because of that, many companies that currently use these technologies for their products, such as Google, Facebook or Tesla, could see their costs reduced by a large margin. Moreover, by improving the performance of Pytorch 1.13.1 in inference, we could deploy deep learning models in other devices that previously couldn't run them, such as mobile and edge devices. In addition, not only there's a substantial economic interest behind our work, but also an environmental side to it: by improving the performance of deep learning frameworks in training and inference, we can reduce the carbon footprints generated during these processes. This is especially important for HPC applications, where the environmental cost of computing is very high. To showcase these results, we programmed several use cases with Python, in a HPC environment offered by the Barcelona Supercomputing Center (BSC). In the end, we achieved a 10% improvement in performance, and created simple guidelines to outperform 78% of Pytorch's configurations.

Resum

En aquest treball hem millorat el rendiment de la llibreria Pytorch 1.13.1 en aplicacions de High-Performance Computing (HPC) per Unitats de Processament Central (CPUs). El marc de treball de Pytorch és el d'una llibreria de codi obert que té com a objectiu alleugerir l'esforç que suposa programar reds neuronals per aplicacions d'aprenentatge automàtic. Des de la seva creació per part de Facebook, Pytroch ha oferit configuracions per paral·lelisme. Tot i això, aquestes opcions són fixes, la qual cosa implica que no poden canviar durant els processos d'entrenament i inferència de les reds neuronals. En aquests processos, algunes seccions de la red presenten més tasques potencialment paral·lelitzables que altres. Així doncs, si els paràmetres de paral·lelisme que ofereix Pytroch només es poden seleccionar d'una manera fixa, el programa no oferirà el seu millor rendiment. Una possible solució a això seria poder canviar de forma dinàmica les configuracions de paralelisme de Pytorch segons la naturalesa de cada capa de la red neuronal en la que es treballa. Per demostrar que això és possible hem seleccionat la xarxa neuronal LSTM, que conté diverses amplades. En altres paraules, hem escollit una xarxa neuronal que contené seccions en que moltes operacions poden córrer en paral·lel i d'altres en que o bé poques o bé cap poden. A més a més, aquesta xarxa neuronal és actualment la millor per a tasques de NLP (Processament de Llenguatge Natural), la qual cosa la converteix en la red perfecta per a estudiar en aplicacions de HPC. Algunes d'aquestes aplicacions inclouen la traducció, la generació de textos i la predicció de text. En l'estudi actual, hem desenvolupat un cas d'ús en que la red neuronal és emprada per a diversos processos d'inferència. En aquests casos, l'entrenament i la validació de la red ha corregut en tres configuracions diferents: sense paralelisme explícit, amb paràmetres fixes de paral·lelisme i amb paral·lelisme dinàmic. L'objectiu d'aquest treball és demostrar que si s'empren tècniques de paralelisme dinàmic, moltes aplicacions de Pytorch tindran una gran millora de rendiment. A causa d'això, moltes empreses que actualment utilitzen aquests tecnologies en els seus productes, com ara Google, Facebook i Tesla, podrien veure els seus costos considerablement reduïts. A més a més, millorant el rendiment de Pytorch 1.13.1 en inferència, molts models d'aprenentatge profund podran córrer en aparells que fins ara no hi podien córrer, com ara mòbils i rellotges intel·ligents. No només hi ha un substancial interès econòmic darrera el nostre treball, sinó que també hi ha una vessant ecològica, ja que millorant el rendiment podem reduïr en gran mesura la seva petjada medioambiental. Això és especialment important per a aplicacions de HPC, ja que en aquests casos el cost ambiental és molt alt. Per tal de demostrar aquests resultats, hem programat un cas d'ús amb Python, en un entorn de HPC proporcionat pel Barcelona Supercomputing Center (BSC).

Table of contents

ABSTRACT I

RESUM II

TABLE OF CONTENTS III

LIST OF TABLES IV

LIST OF FIGURES V

LIST OF ABBREVIATIONS / GLOSSARY VI

1. INTRODUCTION 1

 1.1 OBJECT 1

 1.2 SCOPE 1
 1.3 REQUIREMENTS 1
 1.4 RATIONALE ..... 1 - 6

2. BACKGROUND AND/OR REVIEW OF THE STATE OF THE ART ..... 6 - 17

3. METHODOLOGY ..... 17 - 20

4. EVALUATING AN ALTERNATIVE SOLUTION ..... ERROR! BOOKMARK NOT DEFINED. - 21

5. RESULTS AND DISCUSSION 22-28

6. BUDGET SUMMARY AND/OR ECONOMIC FEASIBILITY STUDY 28

7. ANALYSIS AND ASSESSMENT OF ENVIRONMENTAL AND SOCIAL IMPLICATIONS 29-30

8. CONCLUSIONS 31

9. REFERENCES ..... 31 -33

List of tables

Table 1 – Runtime results of the LSTM use case without any threads enabled. The first run is considered as a warmup run, used to warm up the computer for the experiments. 22

Table 2 – Statistical analysis of the runtime results from table 1. The analysis contains information about the mean time (s), standard deviation (s), coefficient of variation and margin of error (s). 22

Table 3 – Runtime results of the LSTM use case with a forked section. The first run is considered as a warmup run, used to warm up the computer used for the experiments...23

Table 4 – Statistical analysis of the runtime results from table 3. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s). 23

Table 5 – Runtime results of the LSTM use case with two forks and no thread settings. The first run is considered as a warmup run, a run that is used to warm up the computer used for the experiments. 24

Table 6 – Statistical analysis of the runtime results from table 5. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s)..24

Table 7 – Runtime results for the LSTM use case with two forked sections using different thread settings. The X axis corresponds to intra-operator settings, from 1 to 64 intra-operator threads. The Y axis corresponds to inter-operator settings, from 1 to 16 inter-operator pools or threads. The green cell is the best execution runtime........................24

Table 8 – This table displays the normalization of the results from table 7 to that of the cell with one inter-operator pool and one intra-operator thread (238,72 seconds). The highlighted value is the point of best performance..25

Table 9 – Runtime results of the LSTM use case with a forked section. The first run is considered as a warmup run, used to warm up the computer used for the experiments...26

Table 10 – Statistical analysis of the runtime results from table 9. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s)..26

Table 11 - Runtime results for the LSTM use case with two forked sections using different thread settings and enabling our custom dynamic threading program. The X axis corresponds to intra-operator settings, from 1 to 64 intra-operator threads. The Y axis corresponds to inter-operator settings, from 1 to 16 inter-operator pools or threads. The green cell corresponds to the execution time of the setting with best performance..........26

Table 12 - This table displays the normalization of the results from table 9 to that of the cell with one inter-operator pool and one intra-operator thread (244,96 seconds). The highlighted value is the point of best performance..27

Table 13 – This table shows the percentage difference between the best execution runtimes of experiments nº3 and nº4. The maximum and minimum differences correspond to the highest and lowest possible runtime differences respectively, taking into account the standard deviation of each experiment. 27

Table 14 – This table showcases the normalized results from table 8 with those performances higher than 7,44 highlighted in green. The selection of 7,44 corresponds to the performance of the configuration with the most threads in the fourth experiment.........28

Table 13 – Yearly personals costs working at the BSC..28

List of figures

Figure 1 - Examples of Synchronous Scheduling (a), Asynchronous Scheduling (b), and changing from one to four thread pools using the same quantity of hardware resources (CPUs)(c). 3

Figure 2 – Performance of the visual NN Inception v2 with different number of inter-op pools and intra-op threads (MKL Threads) per pool. The The performance is normalized to that of one pool with one intra-op thread. The best configuration is the one balancing intra-op and inter-op parallelism. (Source [5]) 4

Figure 3 - Execution traces of a neural network with layers of variable width. (Source [5]) ...5

Figure 4 – Table with the main observations drawn from their study regarding different hardware and software platforms (source [6]) 7

Figure 5 – Graph detailing different execution times for several DL models, using the SMAUG program. It is an important advance the detailing of the different execution parts. (Source [7]) 8

Figure 6 – Normalized time to train the CosmoFlow benchmark for different HPC hardware platforms. (Source [9]) 9

Figure 7 – Per iteration latency on 16 GPUs (top row) and 32 GPUs (bottom row). Different hardware platforms are trained on vision and language models and their latencies per iteration are compared. Models such as ResNet50 present an almost linear scalability (b). (Source[11]) 10

Figure 8 – PyTorch: Multi-Node performance of ResNet and Inception models using the AMD EPYC hardware platform. This example of results shows an almost linear treend between number of nodes and number of images per second the hardware is able to process. 12

Figure 9 – This diagram shows how inter-op threading is performed in Pytorch. Forking is used to enable asynchronous threading. (Source [15]) 13

Figure 10 – This figure is an example code of how forking is performed in Pytorch. In this case, the task of matrix multiplication runs asynchronously. 14

Figure 11 – Example of basic forking to run a function in Pytorch asynchronously. 15

Figure 12 – A loop inside a Pytorch's forked function. 16

Figure 13 – Display of the execution traces of 78 CPUs using Intel's Pytorch extension for profiling. 17

List of abbreviations / Glossary

High-Performance Computing → HPC

Central Processing Unit → CPU

Machine Learning → ML

Long Short-Term Memory → LSTM

Natural Language Processing → NLP

Barcelona Supercomputing Center → BSC

Federated Learning → FL

Graphic Processing Unit → GPU

Tensor Processing Unit → TPU

Deep Learning → DL

Amazon Web Services → AWS

1. Introduction

1.1 Object

The objective of this project is to prove that we can improve the performance of the Python library Pytorch 1.13.1 for CPUs in situations where neural networks can be parallelized.

1.2 Scope

A use case has been developed to provide empirical results of our improvements in performance. This use cases consist of the LSTM neural network for language processing. The network has been developed to reflect actual applications and has been programmed with Python, the default language for Pytorch.

In terms of the work performed, the entirety of this project has been done as part of my position working at the Barcelona Supercomputing Center as a Master Student. In my position I was led by professor Vicenç Beltran and worked in collaboration with Aleix Roca. All the work and results you will see in this project has been done exclusively by me, with the help and guidance of both BSC researchers. Nonetheless, I will be using the *we* pronoun instead of *I* in this study because I couldn't have fully achieved everything without their help, as well as the help from my supervisors Bernardo Morcego / Ramon Pérez. Because of that, on the computing side of the project, we've had the privilege of using the MareNostrum supercomputer, provided by the Barcelona Supercomputing Center. Several computing clusters were employed to run the use cases and to analyse the threading and run of the programs. These processes have been conducted through a Linux kernel and a Gitlab working space.

1.3 Requirements

The use case has been coded in Python, the default programming language for Pytorch 20.12. To test the different applications, we added our newly designed library that optimizes parallel computing. In order to compute these programs, we had to use computing clusters from the MareNostrum supercomputer at the Barcelona Supercomputing Center. The two main limitations were the limited computing resources and the unavailability of online connections while computing. The Barcelona Supercomputing Center assigns workloads to different computing queues, and the only queue we had available contained a maximum of 64 CPUs per workload. Hence, the design of our experiments was made according to this computational constraint. From the online computing perspective, we had to save all files in a local directory. Regardless, all the experiments we pursued are sufficient to prove the objective of our work.

1.4 Rationale

In the last decade there's been a steep increase in the interest towards machine learning. This change in attention is due to three different aspects that now make machine learning more appealing. These new developments are: (i) the creation of improved algorithms that can now be applied to a wider range of areas; (ii) the exponential increase in availability of big data for STEM areas as well as affordable resources to store and analyse it; (iii) the emergence of new hardware architectures capable of handling these processes with the use of parallelism. All these advances allow new algorithms to run efficiently with big data inputs and in a reasonable amount of time [1].

In combination with methods and architectures of HPC, the field of machine learning has seen a notable increase in financing and research. Some of the problems for which machine learning has become a key actor include image analysis, speech, text recognition, and natural language processing, translation, and generation [2]. Because of these recent developments, current machine learning models are more complex than ever, which increases the performance cost of inference.

Clearly, performance is paramount in all machine learning branches, including deep learning. When a deep learning model has finished training, and moves to the deployment stage, any possible improvements in performance have a positive direct effect in the efficiency of the data center resources, since inference is highly sensitive to both latency and throughput [3]. Moreover, the performance of inference also dictates whether or not a model can be deployed to certain devices, such as mobile and edge.

To ease the programmability and interpretability burden of machine learning programs, there's also recently been a considerable effort to develop software frameworks capable of automatically performing some of the hardest tasks. These frameworks have most of their details under the hood while also offering the best performing implementations. Examples of these frameworks are Tensorflow, Caffe, Keras and Pytorch. All of them have specialization in neural networks, one of the most use machine learning techniques, and they also offer features that enable parallelism. Another trait they share is that most of the details running these libraries are hidden to the end user, and only the performance for both training and inference can be easily accessed with timers.

In previous work, the vast majority of machine learning framework research was focused on comparing different frameworks [4]. Even though it is useful information for engineers and researchers that seek to understand the advantages and disadvantages of certain frameworks, there hasn't been any serious effort to characterize the performance of each framework in terms of the different feature choices. Within these features, there's plenty of opportunities for parallelism both within and outside a DL model. The different feature choices available to users will be presented in this study because they are crucial to improve the performance of Pytorch 1.13.1. This selection of features involves the mathematical back-end kernels, the threading libraries, and the scheduling policies.

Currently, Pytorch 1.13.1 is one of the most sophisticated and best performing frameworks for neural network development in both training and inference. Pytorch executes its programs using a dataflow approach in which the end user develops a graph with nodes and edges. The nodes of the graph are mathematical operations like matrix transformations and multiplications, while the edges represent the dataflow dependencies of the tensors. The nodes are also called operators, and their dependencies dictate how much concurrency there is in the operations, in an external sense. Also, operations running inside the operators can be parallelized to improve performance.

In terms of parallelism, Pytorch allows for the selection of a static mode, which can only be set before running the program. This parallelism has two levels, named *inter-node parallelism* and *intra-node parallelism* or also *inter-operator* and *intra-operator* threading. The first type refers to how many operations can be run in parallel at a certain point of execution if the dataflow dependencies are respected, while the second one refers to the amount of parallelism possible inside each operator. Even though these two levels exist, it must be remarked again that the selection of such features is static and can only be done before running a graph-based network.

Before jumping into the different features that Pytorch offers for parallelism, we need to clarify the terms of intra-parallelism and inter-parallelism. Intra-parallelism is the parallelism within an operator or node. The function of operators is to do calculations with tensors, i.e, n-dimensional arrays, and within them there are many opportunities for parallelism. Inter-operator parallelism is the parallelism between operators, which can be exploited by scheduling the neural network asynchronously. We'll talk about asynchronous scheduling later in this section.

Now that we have a clearer understanding of the two main levels of parallelism within

Pytorch, we can move on to the different design features available. The Pytorch framework works with opaque operators, which means that they are manually written and tuned by the end user. In these types of frameworks, there are usually five design features that must be selected prior to executing the network, which are:

Scheduling Mechanism. It's the mechanism in charge of scheduling the operators according to graph dependencies and the availability of hardware resources. The two main mechanisms are synchronous and asynchronous scheduling. The first method schedules each operator at a time, whereas the second one puts all operators in an available state so that they can run when computing resources are available. In the example from Figure 1, asynchronous scheduling would be faster if there were unlimited hardware resources. This is because all the tasks from the NN could be done in parallel without any synchronization times. However, considering limited hardware resources, there needs to be a balance between asynchronous and synchronous scheduling. In the scheduling of neural networks there's plenty of potential for inter-operator parallelism.

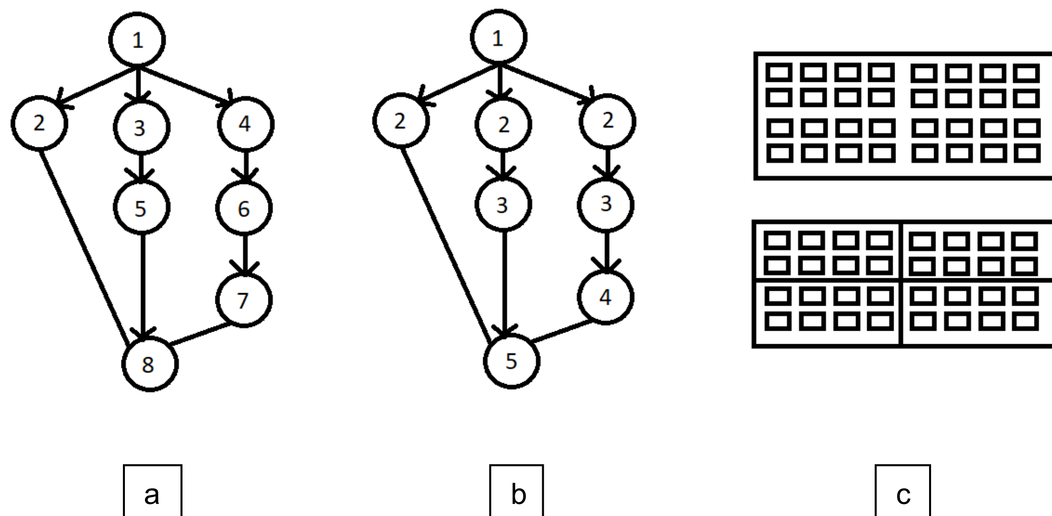

Figure 4 - Examples of Synchronous Scheduling (a), Asynchronous Scheduling (b), and changing from one to four thread pools using the same quantity of hardware resources (CPUs)(c).

Operator Design. There are different ways in which operators can make use of kernels, which contributes to a variety of improvements in performance. A kernel is a central module of an operating system that has complete control over all the other system resources. In this scenario, the kernel is operated by the operator's library and is responsible for managing system calls, interrupts, and other low-level tasks. The kernel acts as an intermediary between the hardware and the software of a computer system. In this study we will not focus on the design of those kernels since it has already been explored in previous research [5].

Math Library. These provide different implementations of kernels, allowing for certain degrees of parallelism to be possible. Moreover, they also provide specific improvements for matrix multiplication.

Thread Pool Library. It refers to the library in charge administering the thread pools. A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work. These threads can be used to perform tasks concurrently, in parallel, without the overhead of creating and destroying threads each time new work is needed.

The basic idea behind a thread pool is that it contains a fixed number of worker threads that are created at the start of the program. When a task is submitted to the thread pool, a

worker thread is taken from the pool and assigned the task. Once the task is complete, the worker thread is returned to the pool so it can be reused for another task.

Thread pools are useful in a variety of situations, but are particularly useful when there are a large number of short tasks that need to be performed. Creating a new thread for each task would be inefficient, because of the overhead of creating and destroying threads. A thread pool, on the other hand, allows you to reuse a fixed number of worker threads, which can help to improve the performance of your program.

Parallelism Mechanism. It is the different set of approaches taken to parallelize the network. It regulates aspects about inter-parallelism and intra-parallelism. For instance, it regulates the number of asynchronous scheduling thread pools, a part of inter-operator parallelism.

Previous work [5] has shown what are the optimal design choices for the recently mentioned features and has also concluded that only one of them needs to change depending on the architecture of the neural network. This feature is the number of asynchronous scheduling thread pools or inter-op pools (*p*). In Figure 1, the number of hardware resources is 32 CPUs, and the number of thread pools is 4, with 8 CPUs per thread pool. In the paper called *Exploiting Parallelism Opportunities with Deep Learning Frameworks [5]* by Wang, Yu Emma, et al., the authors arrived at the conclusion that to achieve the best levels of performance in the Pytorch framework, the number of inter-op pools should be chosen to match the average width of the neural network. In the example from Figure 1, the average width of the neural network architecture represented in the (a) and (b) drawings is 2, assuming all operators perform a similarly computing intensive task. This is because there are 8 total operators in 4 different layers, hence an average width of 2 operators per layer. In the previously mentioned paper, they performed experiments with a neural network architecture with both linear sections and highly branched sections. In other words, there were sections with one node per layer and sections with many nodes per layer. In Figure 2 you can see the results of this experiment over a neural network architecture with an average width of 2.

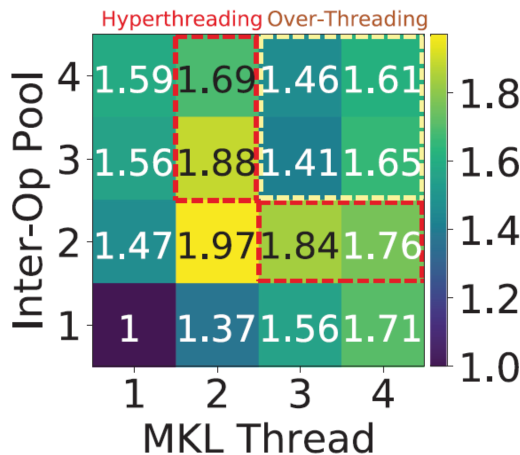

Figure 5 – Performance of the visual NN Inception v2 with different number of inter-op pools and intra-op threads (MKL Threads) per pool. The performance is normalized to that of one pool with one intra-op thread. The best configuration is the one balancing intra-op and inter-op parallelism. (Source [5])

Figure 2 displays the performance of the NN Inception v2 relative to an execution without parallelism, which corresponds to the cell at the bottom left corner. The total number of threads used to run the network is the product of the number of MKL Threads and Inter-Op Pools, where MKL Threads refers to intra-operator threads created with the MKL library. On

the one hand, it is considered hyperthreading when the system runs with more than 4 total threads. On the other hand, when more than eight threads are exceeded its considered over-threading, because the software threads outnumber the hardware threads. We can see that the best performance is achieved with two inter-op pools and two MKL Threads, which is a point where there's a balance between hardware resources and software threads. These results seem counter-intuitive because it would be logic to think that more threads yield better performance due to a higher level of parallelism resources. However, we will see in Figure 3 the reasons why this is not the case.

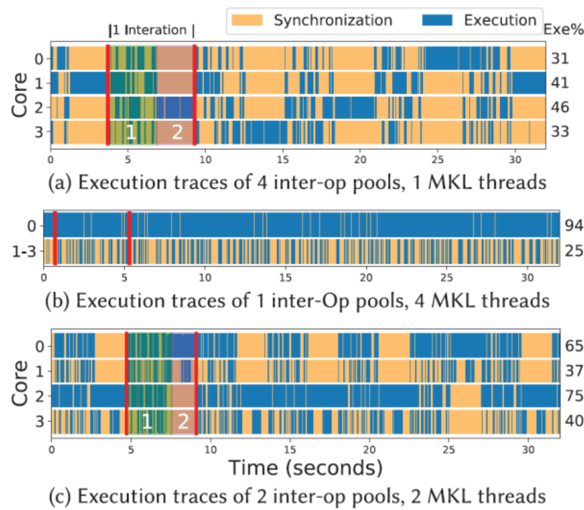

Figure 6 - Execution traces of a neural network with layers of variable width. (Source [5])

In Figure 3, each iteration of the NN is outlined with red bars. The number one is placed at the bottom of a section with a high network layer width, whereas the number two is placed at the bottom of a section running with a width of one operator per layer. Hence, the areas with a 1 show both intra-parallelism and inter-parallelism while the areas with a 2 display only intra-parallelism. At the right side of each trace, it is displayed how much time each core spends executing versus synchronizing. In the first case (a) where there are 4 inter-op pools and just one MKL thread there's a lot of synchronization overhead because the operators from section 2 have only one core assigned to execute while the other three are waiting. The second case (b) doesn't present good levels of performance either because the native operations of the framework, the operations happening inside the operators, are mostly single-threaded. Because of that, the CPUs running the framework's native library thread will take more time to run than the others, which will result in high synchronization times. While that's happening, the other CPUs won't have any work available. Finally, the third case provides the best of both worlds by reducing synchronization times compared to cases (a) and (b). This is because it can both use the benefits of inter-parallelism and intra-parallelism when the network is wide and the benefits of intra-parallelism when the network has only one operator per layer.

After seeing these results, it becomes clear that there's still room for improvement. There are many neural network architectures consisting of different dependencies and operator designs. If, according to the results of the paper we've just mentioned, we fix each thread pool size in a fixed way, most NN will suffer from synchronization overhead due to imbalances in the computing efforts required by different parts of the network. Hence, there's the opportunity of doing this process dynamically by providing different threading resources to different operators. A good example is in Figure 3, where there would be a substantial improvement in performance if two inter-op pools and two intra-op threads were assigned in Area 1, while Area 2 had one pool of four intra-op threads. It must be mentioned that these results apply for both inference and training.

In the past, there have been experiments made to investigate which are the parallelism features most suitable for certain machine learning applications and NN architectures. These studies have shown that there is a significant margin of improvement in performance for most use cases in Pytorch, in terms of parallelism. The pursuit of dynamically changing parallelism features during the run of a program, also called malleability, has been already researched in the field of linear algebra, with outstanding improvements in terms of performance, flexibility, and usage of resources. In the field of Machine Learning, and more specifically to Pytorch, a completely malleable implementation unlocks the option of dynamically changing the parallelism configurations while a program is being executed. Due to the huge optimization opportunity presented in previous research for these frameworks, in this study we will explore how to change parallelism features in a dynamic way and will also present the experimental results.

Our initial hypothesis is that a program capable of o dynamically changing an assigning inter-operator and intra-operator threads will perform better than Pytorch's fixed threading implementation, both in training and inference.

2. Background and/or review of the state of the art

As recently mentioned, most of the previous research in DL frameworks had focused on cross-framework comparisons instead of researching the inner configurations of each framework. The first study to seriously research different frameworks in depth is the paper called *Exploiting Parallelism Opportunities with Deep Learning Frameworks* by Wang, Yu Emma, et al. *[5].* In that paper, several frameworks were studied in depth, while also offering ideas for potential optimizations. Our study is focused in one of these ideas, related to the balance between intra-op and inter-op parallelism within neural networks, both for inference and training. In this review of the state-of-the-art we will not go over this paper again, as it has been extensively discussed already. The focus of this review will be now the several papers that emerged after that one, and what they have offered to our research. Hence, this section will be divided into nine different subsections, each one of them detailing a different paper, along with its relationship with our proposed research.

2.1 A systematic methodology for analysis of deep learning hardware and software platforms [6] by Wang, Yu, Gu-Yeon Wei, and David Brooks

One of the most important aspects of developing new deep learning improvements and optimizations is to be able to perform a systematic comparison with a standardized approach. This paper offers exactly that, a pipeline that allows a comparison between deep-learning techniques. The publication contains a set of hardware and software tools that ease the burden of creating a framework to compare deep learning methodologies. Until the publication of this paper, there were no clear and universal techniques to perform this type of comparisons. Moreover, their methods can be used alongside with traditional use cases.

The rapid development of both the deep learning models and the hardware platforms requires a systematic method to shine a light towards interactions of the two and the different features of the model. By developing such a standardized approach, the resulting insights can be promptly used in future work. One of the main drawbacks of the previously used benchmarks was the slow development cycle of their platforms, as well as their limited size. The neural network models that were being included in the benchmarks seemed arbitrary and didn't truly reflect the most important attributes of a DL model. In addition, the models that were being included became outdated in a few years and it took an extensive amount of time to add new ones. Moreover, there are very substantial differences when running the same DL model on different hardware platforms, which could lead to misleading

conclusions.

The systematic analysis proposed in this paper is a comprehensive performance evaluation methodology that includes benchmarks with parametrized DL together with techniques for systematic analysis. The work extends over the current benchmarks while short-cutting some of the unnecessary applications, which didn't include fundamental information about the performance of NNs.

In regards to the conclusions drawn in this study, we used this methodology in our work as a reference point to devise an evaluation strategy for the different use cases. There were several insights presented in this paper that weren't included in previous research. These insights offered us a way to both expand the range of our benchmarks and avoid the mistake of not fully evaluating the models to the extent offered by the hardware platforms that were at hand. All in all, this research settled the foundations from which to build a reproducible and meaningful study

Observation	Proof	Insight/Explanation
1. TPU exploits the parallelism from batch size and the model width.	Fig 2	To design/upgrade new specialized systems, architects need to consider interactions between the operation mix from key workloads (arithmetic intensity) and system configurations (FLOPS, memory bandwidth/capacity, intra-chip/host-device interconnect). TPU serves as a great example.
2. Many operations are bottlenecked by TPU memory bandwidth.	Fig 3	
3. TPU suffers from large inter-chip communication overhead.	Fig 4	
4. Smaller CNN models are more bottlenecked by CPU hosts.	Fig 5	
5. TPU v3 speeds up compute-bound MatMuls by 2.3×, memory-bound ones by 3×, and large embeddings by > 3×.	Fig 6	
6. The largest FC models prefer CPU due to memory constraints.	Fig 7	Need for model parallelism on GPU and TPU.
7. Models with large batch size prefer TPU. Those with small batch size prefer GPU.	Fig 8 Fig 10	Large batches pack well on systolic arrays; warp scheduling is flexible for small batches.
8. Smaller FC models prefer TPU and larger FC models prefer GPU.	Fig 8	FC needs more memory bandwidth per core (GPU).
9. TPU speedup over GPU increases with larger CNNs.	Fig 10	TPU architecture is highly optimized for large CNNs.
10. TPU achieves up to 3× FLOPS utilization compared to GPU.	Fig 11	TPU is optimized for both CNN and RNN models.
11. GPU performance scales better with RNN embedding size than TPU.	Fig 10	GPU is more flexible to parallelize non-MatMuls.
12. Within seven months, the software stack specialized for TPU was improved by up to 2.5× (CNN), 7× (FC), and 9.7× (RNN).	Fig 12	It is easier to optimize for certain models than to benefit all models at once.
13. Quantization from 32 bits to 16 bits significantly improves TPU and GPU performance.	Fig 12	Smaller data types save memory traffic and enable larger batch sizes, resulting in super-linear speedups.
14. TensorFlow and CUDA teams provide substantial performance improvements in each update.	Fig 12	There is huge potential to optimize compilers even after the hardware has been shipped.

Figure 4 – Table with the main observations drawn from their study regarding different hardware and software platforms (source [6]).

.

2.2 SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads by Xi, Sam, et al. [7]

These past years, the field of deep learning has seen a very steep advance in hardware infrastructure and hardware accelerators, which has led to a substantial improvement in performance. Past research has largely focused on improving the microarchitecture of accelerators, with the objective of reducing energy consumption an improving performance at each layer of the network. The most revealing part of this study is the finding that design of accelerators accounts for only 25 – 40% of the inference latency, whereas the other 60% relies exclusively on the flow of data and the design of the framework. Up until this research and the previously mentioned study, reviewing the performance of neural networks in a standardized end-to-end approach was very difficult. The main reason for that was the lack simulations of DNN frameworks including hardware accelerators. Hence, SMAUG was devised to exactly fill this demand.

In their work, the authors present a case study in which they are able to improve the performance of a deep learning network by 1.8x only by applying changes to the dataflow dependencies and the framework settings, without modifying anything about the accelerator's microarchitecture. SMAUG is also able to automatically tune deep learning settings according to the demands of the use case. In regard to our project, SMAUG has been incredibly helpful in adjusting certain parameters that ended up improving the performance of our use case. For instance, it brought up to our attention that by changing

the dataflow dependencies we could see a substantial improvement in the pre-processing time for our data. This work builds upon existing literature to deliver a solid starting point from where to refer our improvements, without sacrificing any information about accelerators.

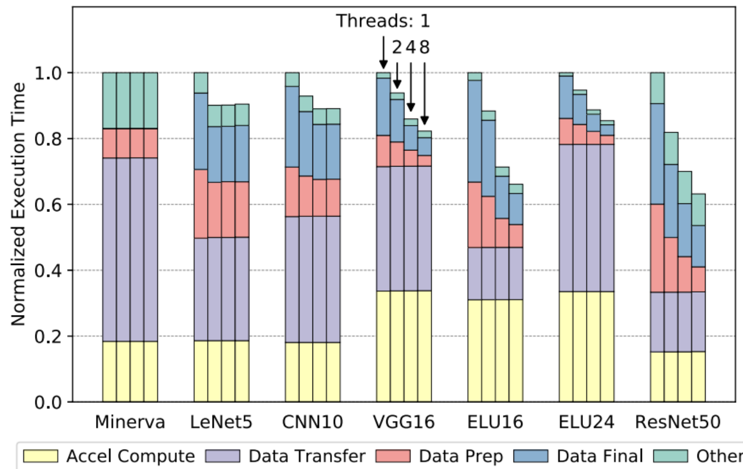

Figure 5 – Graph detailing different execution times for several DL models, using the SMAUG program. It is an important advance the detailing of the different execution parts. (Source [7])

2.3 A survey of deep learning on CPUs: Opportunities and Co-Optimizations by Mittal, Sparsh, Poonam Rajput, and Sreenivas Subramoney [8]

This study surveys the advantages and disadvantages of CPUs for DL applications. It also explains what are the current opportunities for improvement and utilization. The paper concludes that CPUs are a crucial tool for running DL applications on all sorts of devices, from edge to supecomputers. One of the main advantages that the authors see in CPUs over other accelerators is their standarization and portability. To argue their points, they show how CPUs are proficiently understood by industry and academics alike, with a detailed hardware/software stack. Morevoer, it is clear that CPUs are present in most systems, hence the importance of understanding DL performance on CPU systems. In short, the main reasons why CPUs are important for DL is that they are the most available, software ready and portable pieces of hardware.

The main contribution of the paper is its comparison of CPUs over accelerators. The study details in depth where CPUs perform best and what are some opportunities for further improvement. It also displays what parameters must be tuned in order to achieve best CPU performance compared to using an accelerator. In another really important section of the research, there's a thoroughly detailed explanation of how a DL model's architecture affects the performance of a CPU compared to the performance of an accelerator. This way, the design of a neural network can be modified to meet the requirements of a CPU, potentially allowing it to run on a wider range of devices at a lower cost. The work includes studies performed in a wide range of devices, from mobile and edge to cluster and server. Finally, the investigation also provides examples of memory usage in the different use cases provided.

This piece of literature was crucial to our work because it laid out the fundamental literature to justify continuing work with CPUs, as started by the original paper [5]. Because of the different points explained in S. Mittal's [8] work, it is clear that by further researching the optimization opportunities presented by CPUs, the world of DL will be substantially benefited.

2.4 MLPerf™ HPC: A holistic benchmark suite for scientific machine learning on HPC systems by Farrell, Steven, et al. [9]

Once again, a paper appears with the goal of creating a systematic benchmark for DL models. However, this time its goal is to provide an adaptation to HPC applications. Their main motivation is similar to the one already explained in previous papers: the steep adoption of DL in many areas of academy and industry requires for a standardized approach to compare models and optimizations. But, as we already explained, in this study the focus shifts to HPC applications. The community that most uses HPC is the scientific. Their computing systems are at the edge of performance, with a plethora of hardware resources available to them, as well as capabilities that heavily rely on scaling. Because of that, the recent developments in ML and DL need to be proven to work with this type of resources and they must be shown to work at scale. With that objective in mind, MLPerf^{TM} comes in as a benchmark of scientific use cases representative to real-world scenarios. Their workloads contain three different methods of analysis based on either the dataflow, the design of the neural network and the performance. The fact that so much information is contained in the analysis enables both a qualitative and quantitative understanding of scaling, compute-unit utilization, communication scheduling and potential optimizations. Such a complete picture of DL performance on HPC systems wasn't present in previous studies. Some of the most relevant information present in this research contains the scale-dependent influences of dataset size, together with the HPC system's memory and training/inference latencies.

The authors propose several different techniques to overcome certain scalability challenges related to large batch-sizes. However, the most important aspect of this research is that they have also included the results of several different scientific supercomputers world-wide in their proposed DL benchmarks. This kind of information wasn't present in previous studies and paves the way towards standardised HPC benchmarking of DL applications. For our work, this vital information is very useful to us as it serves as a rooftop for performance. It is very difficult to gather information about other supercomputers as they are very difficult to access and most of the information output usually goes through several layers of bureaucracy. By relying on this piece of literature, we could now make a more educated guess about the progress of our work, in comparison with other supercomputers worldwide.

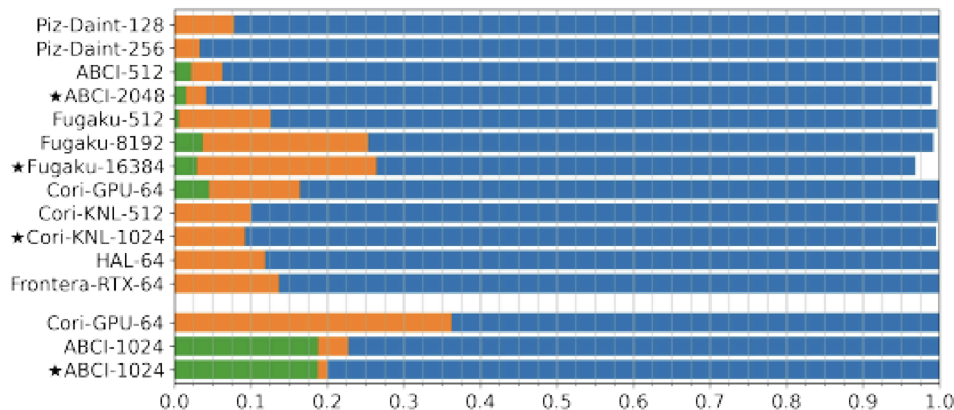

Figure 6 – Normalized time to train the CosmoFlow benchmark for different HPC hardware platforms. (Source [9])

2.5 AutoFL: Enabling Heterogeneity-Aware Energy Efficient Federated Learning by Kim, Young Geun, and Carole-Jean Wu [10]

This paper improves over the most recent techniques of federated learning. Federated learning is a machine learning technique that allows multiple users or devices to train a model on their own data, without sharing that data with each other. This approach can be useful in situations where data is distributed across many devices and centralizing the data for training would be difficult or infeasible. In federated learning, each user or device trains a local version of the model on their own data, and then the trained models are aggregated and averaged to produce a global model. This global model can then be used to make predictions on new data. Federated learning can help preserve the privacy of user's data, as the data never leaves their device and is only used to update the local model. One of the main limitations in enabling the deployment of efficient FL on edge devices is the varying runtimes. In other words, it is difficult to use a federated learning approach when all of the devices have a different performance. In this study they display the current state-of-the-art for FL, while also sharing the current limitations and opportunities for optimization.

Federated learning is a key component for running DL on edge and mobile devices. Therefore, it was crucial to our work to understand which are the current limitations and possible improvement opportunities. After understanding this pioneering piece of literature, it was clear to us that the advancements in performance that we intended to deliver with our work would clearly help some of the challenging aspects of FL. In addition, most of the key concepts about running DL models on edge and mobile devices are present in this article, useful information that helped us steer the focus of our investigation.

2.6 Pytorch Distributed: Experiences on Accelerating Data Parallel Training by Li, Shen, et al. [11]

This paper is a showcase of the distributed data parallel module from Pytorch. The input data for DL training is exponentially growing, and so is the demand for scaling this process to higher computational capacities. One potential answer to this demand could be the use of data parallelism, a new popular solution to distribute training. In simple terms, the data parallelism approach creates a replica of a DL model in each computational unit (CPU, GPU

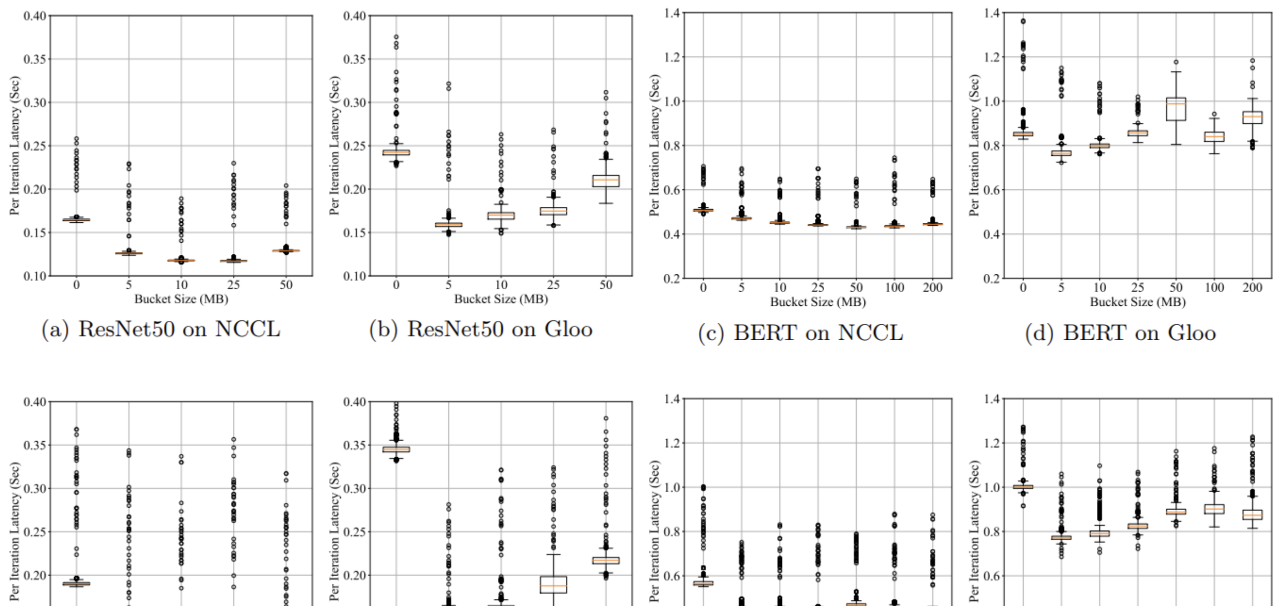

Figure 7 – Per iteration latency on 16 GPUs (top row) and 32 GPUs (bottom row). Different hardware platforms are trained on vision and language models and their latencies per iteration are compared. Models such as ResNet50 present an almost linear scalability (b). (Source[11])

or TPU) and generates independent gradients. After that, for each iteration these gradients are shared between units for consistency. Even though this concept seems simple, it has been proven to work well for large amounts of data, while also providing excellent scalability properties. In recent updates, the Pytorch library has also included some features to accelerate data parallel training, which are evaluated in this paper. This study concludes that when data parallelism is appropriately used in Pytorch, it can achieve near-linear scalability. This research performed by Li, Shen, et al. is very relevant because prior to it there were no studies that could demonstrate linear scalability of a DL framework, since all showed an exponential trend.

In terms of our own study, the results of this paper are very interesting because data parallelism could be used in combination with what we propose, in future work. To our present research, data parallelism serves mostly to compare our results with another technique devised to improve performance in HPC applications. In that regard, the data parallel training module from Pytorch proposes a new way forward in terms of training big datasets, and we are interested in understanding how our program can compete. With that objective in mind, the paper includes enough data to confidently perform a quantitative and qualitative comparison.

2.7 Performance Characterization of DNN Training using Tensorflow and Pytorch on Modern Clusters by Jain, Arpan, et al. [12]

This paper presented by Jain, Arpan, et al. dives deep into characterizing DL training for CPUs in the Pytorch and Tensorflow frameworks. The main reason behind the study was the lack of academic material focused on CPU training, since most of the research had been done on GPUs. The most interesting part of this study was to finally be able to compare the performance of different CPUs. On top of that, the article also includes key insights about the performances of Tensorflow and Pytorch, according to the use of certain configurations.

On the one hand we've got the comparison of different CPU architectures to state-of-the-art DL networks such as ResNet(s) and Inception-v3/v4. The information presented in this section is vital because it allows us to compare it with the hardware we've used in our experiments. Also, the networks they use to test their CPUs are very useful to our case study and the results they obtained with them could be also used as a starting point for our investigation. Given all of that, the results tables presented in this work were used as a reference to guide and validate our initial findings.

On the other hand, the insights the authors drew from their research are very useful to our project, since they include information on when and how to use Multi-processing (MP) parameters. Then, the paper also researches how these parameters affect CPU performance and what is the relationship between that and variables such as NN architecture, CPU architecture and framework settings. Most interestingly, some of the results presented in this work raise questions that the paper *Exploiting parallelism opportunities with deep learning frameworks [5]* answers. One of these questions is: why is there a non-linear relationship between the characteristics of the system and the CPU (core-count, ppn, hyper-threading, etc.) and DNN specifications like inherent parallelism? The answer is in how the CPUs distribute their execution times between units, which has been well explained in our rationale. The fact that this question was present once again in this investigation shows even more solid reason for our work. Taking everything into consideration, this paper offers information that was very much needed for our project, and reaffirms the motivations of our study.

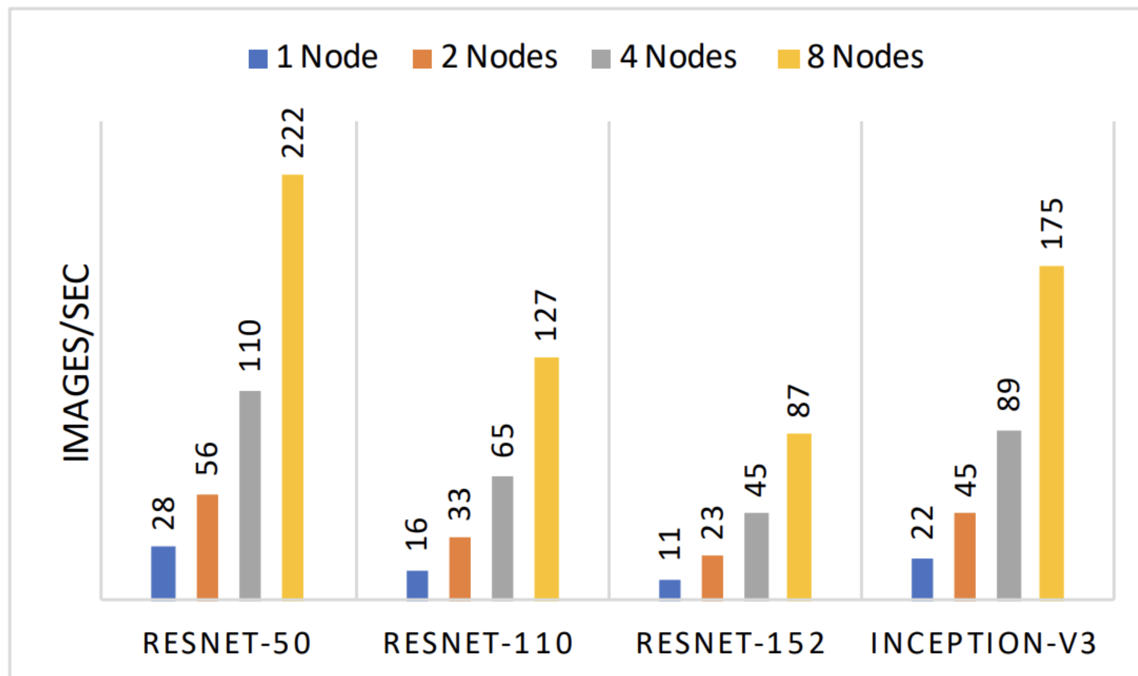

Figure 8 – PyTorch: Multi-Node performance of ResNet and Inception models using the AMD EPYC hardware platform. This example of results shows an almost linear treend between number of nodes and number of images per second the hardware is able to process.

2.8 A linear algebraic approach to model parallelism in Deep Learning by Hewett, Russell J., and Thomas J. Grady II [13]

This study goes back to the fundamentals to create a linear algebraic approach for representing the parallelism opportunities within neural networks. Following the same theme as all previously mentioned papers, it is clear that the increase in size and complexity of DL models goes hand-by-hand with an increase in the computational capacity required to compute them. Hence, the computing environment of DL is shifting more and more towards large clusters in combination with data and model parallelism methods. The goal would be to distribute any tensor of the NN between every parallel computing unit (CPU, GPU or TPU). To achieve that, the authors devised a method of translating data movement operations into linear operators. They attained this objective by defining the relevant spaces and the inner products while also manually developing the operators necessary to perform the gradient-based training of DNNs. By using these parallel primitives, they managed to create highly distributed DNN layers. The utility of their program was showed in a use case.

The objective of our work was to implement a parallelism library that dynamically adjusted the parallel distribution of tensor operations across CPUs in DL applications for training and inference. Hence, this study resembles our goal in the sense that it also tries to increase the parallelism abilities of NNs within the Pytorch framework. Even though the approach they use in this paper does not match our methods, the fundamental reasoning behind their study is the same as the one we used the develop our library: going back to the fundamentals of NN computation to achieve higher levels of parallelism. Therefore, though we propose a technique to go even deeper into these fundamentals, this study serves well to compare how a similar approach performs in training. Some of the insights gained there matched our results, and their weak points and limitations are covered by our approach. In that sense, this work could be seen as an intermediate step to our final solution, and their results as a valid reason to continue in their direction: to go back to the

fundamentals of NN computation to better understand a DL model's inner abilities for parallelism.

2.9 PyTorch: An Imperative Style, High-Performance Deep Learning Library by Paszke, Adam, et al. [14]

Finally, the last piece of literature we used to base our research upon is none other than the Pytorch library documentation itself, the most reliable and up-to-date resource. Some of the information presented in this paper made us choose Pytorch over other DL frameworks such as Tensorflow or Caffe. In the end, we were more inclined to use Pytorch because it's the framework that better balances usability and speed while also being well documented with a substantial community behind it. In addition, debugging in Pytorch is very straightforward and consistent with other scientific computing libraries. We won't go into detail about the inner architectures of Pytorch, however we will present four pieces of documentation that helped us improve our work and move it ahead. The next subsections present these tutorials:

2.9.1 CPU threading and torchscript inference [15]

This official Pytorch tutorial lays out the fundamental practices of creating threads for CPU computing of DL inference.

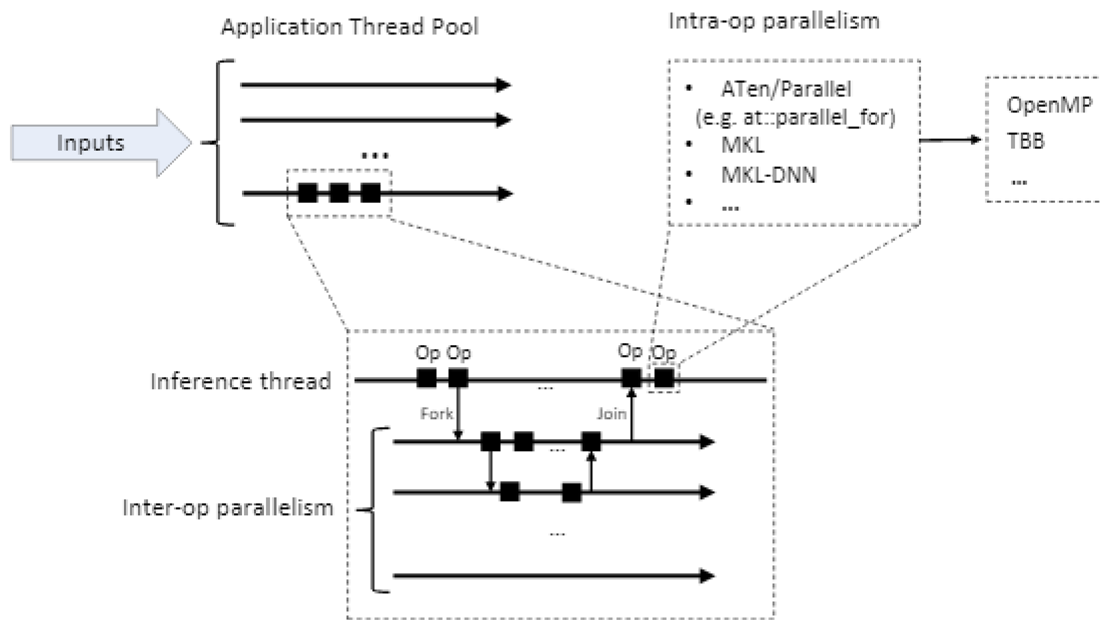

Figure 9 – This diagram shows how inter-op threading is performed in Pytorch. Forking is used to enable asynchronous threading. (Source [15])

There are two main conclusions to be drawn from this tutorial: first, that to be able to run a neural network asynchronously there must be an explicit forking of the code. In other words, the inference has to be explicitly divided in different paths so that each path can be computed by a different pool of CPUs. This process is performed using a *fork* from TorchScript, which returns a future object that can be synchronized later.

```
@torch.jit.script
def compute_z(x):
    return torch.mm(x, self.w_z)

@torch.jit.script
def forward(x):
    # launch compute_z asynchronously:
    fut = torch.jit._fork(compute_z, x)
    # execute the next operation in parallel to compute_z:
    y = torch.mm(x, self.w_y)
    # wait for the result of compute_z:
    z = torch.jit._wait(fut)
    return y + z
```

Figure 10 – This figure is an example code of how forking is performed in Pytorch. In this case, the task of matrix multiplication runs asynchronously.

The second conclusion is that the fundamental ideas of inter-op and intra-op parallelism mentioned in the rationale of this work are also present in Pytorch.

If we want to set the number of intra-op and inter-op threads in our code, we will have to explicitly do so with the following commands:

For inter-op parallelism:

set_num_interop_threads()

For intra-op parallelism:

set_num_threads()

In the same tutorial, the authors from the Pytorch team also make it clear that there needs to be a manual tuning of threads, since adding to many threads will lead to oversubscription. The fact that they admit this limitation in their official tutorial provides even further reason to our work, which has the objective of reducing this oversubscription to a minimum.

2.9.2 Dynamic Parallelism in Torchscript [16]

This official Pytorch tutorial, even though it claims to dynamically parallelize TorchScript applications, we will now see how it doesn't truly fully apply dynamism. Initially, the documentation aims to explain the syntax for performing dynamic inter-op parallelism in TorchScript. This type of parallelism is dependent on the control flow of the program, and touches only inter-op parallelism, which is only concerned with running different sections of a neural network in parallel, but not from within an operator. An example would be the following code snippets:

```
import torch

def foo(x):
    return torch.neg(x)

@torch.jit.script
def example(x):
    # Call `foo` using parallelism:
    # First, we "fork" off a task. This task will run `foo` with argument `x`
    future = torch.jit.fork(foo, x)

    # Call `foo` normally
    x_normal = foo(x)

    # Second, we "wait" on the task. Since the task may be running in
    # parallel, we have to "wait" for its result to become available.
    # Notice that by having lines of code between the "fork()" and "wait()"
    # call for a given Future, we can overlap computations so that they
    # run in parallel.
    x_parallel = torch.jit.wait(future)

    return x_normal, x_parallel

print(example(torch.ones(1))) # (-1., -1.)
```

Figure 11 – Example of basic forking to run a function in Pytorch asynchronously.

```
import torch
from typing import List

def foo(x):
    return torch.neg(x)

@torch.jit.script
def example(x):
    futures : List[torch.jit.Future[torch.Tensor]] = []
    for _ in range(100):
        futures.append(torch.jit.fork(foo, x))

    results = []
    for future in futures:
        results.append(torch.jit.wait(future))

    return torch.sum(torch.stack(results))

print(example(torch.ones([])))
```

Figure 12 – A loop inside a Pytorch's forked function.

We have already seen a very similar piece of code in the Pytorch tutorial we first discussed. The main difference here, however, is their use of a loop to iterate to the number of tasks. It is dynamic in the sense that it depends on the type of inference we are performing, and in the number of iterations, but the selections for inter-op and intra-op parallelism remains fixed throughout the computing of the process. Because of that, even if the level of dynamism increases in respect to the first tutorial, it does not yet reach its full potential, which is what we intend with our work.

2.9.3 Grokking Pytorch Intel CPU Performance from First Principles [17]

In this case, we also studied an unofficial Pytorch tutorial presented by Min Jean Cho and Mark Saroufim. The objective of their work was to present a TorchServe case study, a tool for serving PyTorch models. It is a flexible and easy-to-use tool that makes it simple to deploy PyTorch models at scale. TorchServe also provides support for model versioning, to easily roll back to a previous version of a model if necessary. Additionally, it provides support for automatic batching, which can improve the performance of a model when serving large numbers of requests. Overall, TorchServe is a useful tool for deploying PyTorch models in production environments. The goal of the study was to optimize TorchServe for CPUs using an Intel extension.

The two main problems they were trying to fix were the bottlenecked GEMM execution units and the Non-Uniform Memory Access(NUMA). In DL frameworks, GEMM (General Matrix Multiply) runs on Fused-Multiply-Add (FMA) or Dot-Product (DP) execution units. If hyperthreading is enabled, these units can be bottlenecked in the synchronization of threads, because there's a contention for the same core resources by all working threads. The solution proposed in this study consists in assigning only one thread per physical core by setting them to have CPU thread affinity. The other problem is in regard to memory access. NUMA is defined as a shared memory architecture that characterises multi-socket systems. The architecture describes the placement of main memory modules with respect to processors. One potential problem that Pytorch presents is if the used processors are not NUMA-aware, then memory runs very slow. The tutorial solves this by setting CPU thread affinity to a specific socket using core pinning.

Most of the conclusions drawn on this study were possible through an Intel extension that enables in detail CPU profiling. To our work, this documentation helped us mostly to understand certain configurations of thread affinity and the Pytorch Intel extension for CPU profiling. We couldn't use some of the solutions present in this study because even if they solve some CPU limitations, they were not compatible with what we were trying to apply. In future work, we would like to be able to implement these methods with our new practices. All in all, the tutorial served mainly to understand the architecture and configurations of Pytorch in regards to CPUs more than to improve our code, while also providing crucial tools for profiling.

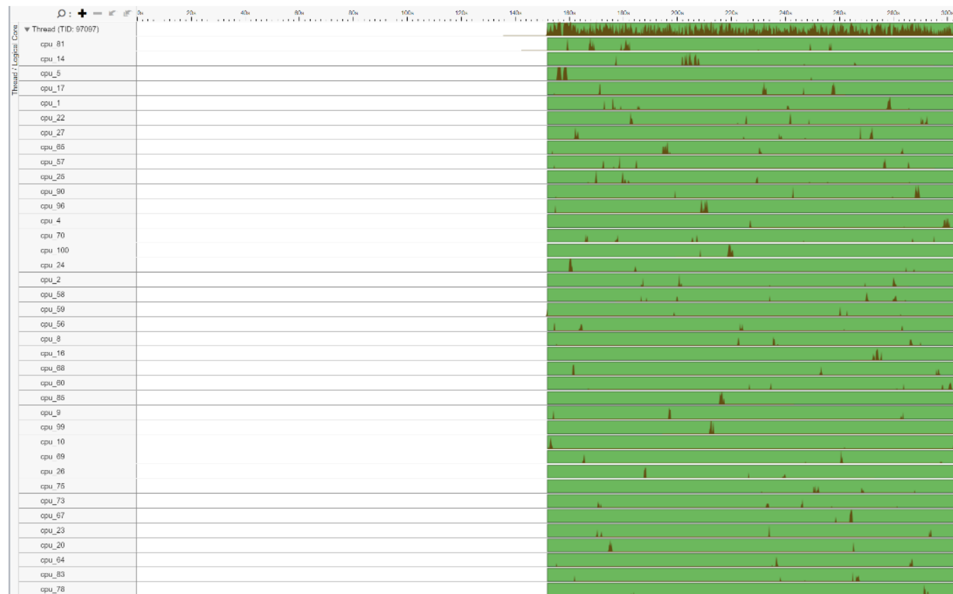

Figure 13 – Display of the execution traces of 78 CPUs using Intel's Pytorch extension for profiling.

2.9.4 Performance Tuning Guide [18]

This last piece of documentation is also an unofficial Pytorch tutorial presented by Szymon Migacz. This performance tuning guide contains a diverse set of optimizations and techniques that improve the performance of Pytorch for training and inference of DL models. Some of these modifications are specifically important for CPUs and proved very useful to our programs. In this section we will not go into detail about each specific technique because it is not the purpose of a literature review. However, we will present all modifications from our code in the annex, where all of our programs will be presented. In addition, we will also create a Github repository containing all our code.

3. Methodology

To prove our initial hypothesis, we had to devise a set of experiments that could unequivocally confirm it. In different sections of this methodology part, we will explore which are these experiments and what are the steps we took to execute and refine them. There are many references to the GitHub[1] repository of this work, where the code of our study is presented.

3.1 Benchmarking DL frameworks

Firstly, we had to create different benchmarks to obtain reference results from which to improve upon. Initially, we started by creating a benchmark to evaluate the Tensorflow 2.0 DL framework. Our testing grounds consisted of training and inference workloads for different NNs. The selection of these networks was made in regards to the paper we've mentioned in the rationale of this work [5]. We were interested in evaluating NNs with a high average width, because these are the ones offering higher intra and inter-operator parallelism opportunities. Because of that, we chose to work with transformers and the inception V3 NNs.

On the one hand, transformers are a type NN mostly used for NLP. They are characterized by having a large average width of about 4 operators per layer. Hence, this makes them the perfect candidate to evaluate parallelism for both training and inference.

Currently, they are one of the best performing NNs and their use is increasing exponentially. On the other hand, there's inception V3, a NN mostly used for vision that has an average width of about 3 operators per layer. Upon further inspection of these two networks, we found out that we were not able to either reproduce the results of the original paper or to change the parallelism settings of Tensorflow at a kernel level. This was because Tensorflow uses its own parallelism kernel, called nsync [19], which is a C library that exports various synchronization primitives. In other words, nsync computes nearly optimal synchronization plans, and it wasn't feasible to us changing that code.

Because of that, we decided to use Pytorch 1.13.1. According to the original paper [5], their results also applied to the Pytorch DL framework.

3.2 Benchmarking Pytorch 1.13.1

Once we had established Pytorch 1.13.1 as our DL framework of choice, we began creating a platform to evaluate NNs in training and inference. In this section, we will go through the two steps we took to benchmark Pytorch 1.13.1 in inference. The results we obtained are also applicable to training, as mentioned in the literature.

3.2.1 Developing a use case for inference

The first step was to create a use case with the necessary features for a possible improvement by our library. More precisely, we first needed to reproduce the results of the paper our work is based on [5]. For that, we used the code provided in the tutorial about Dynamic Parallelism in Torchscript, made by the Pytorch team [16]. The code contained a use case for inference of the LSTM NN, a network mostly used for NLP that has an average width of 2,7 nodes per layer. The only two changes we applied to the code were changing the batch size, the number of time-steps and adding the inter and intra-operator settings, as described in the official Pytorch tutorial [15]. The average width of the network was adequate because it was even bigger than the one used in the article of reference [5]. In the context of machine learning and deep learning, a batch size refers to the number of samples from the training or testing dataset that are used in one iteration. The samples in a batch are processed together by the LSTM network, to update the model's parameters. The choice of batch size has an impact on the performance of the inference. Because of that, we changed the batch size and the number of time-steps of the original settings to 400 each, so that the program could take between 20 and 30 seconds to execute. The reason behind this time constraint is that, in HPC environments, we can't accurately measure the differences in the performance of a program that runs in under 20 seconds.

That is because under that threshold the changes might be attributed to other variables related to the settings of the environment. In parallel, we also enabled the parsing of parallelism variables. The entire code can of this use case can be found under *First_experiment* in our GitHub[1] repository.

3.2.2 Testing the use case in an HPC environment

After developing the use case, we had to make sure that we could reproduce the results of the reference article [5] in an HPC environment. In order to do that, we developed scripts that allowed us to execute the script using 64 physical CPUs. Once again, the code is in the three folders of the GitHub repository, under the names of *run.sh* and *job.sh*. We chose to work with 64 physical CPUs because that's a likely computing scenario for an HPC environment. Moreover, having more CPUs allows for higher parallelism because we can use a higher number of threads, each of them running in a CPU.

---

[1]GitHub Repository: https://github.com/martillopartUPC/Pytorch-Parallelism---TfM.git

3.2.3 Refining the tests

Initially, the parallelism configurations we used to run the different executions of the use case spanned from 1 to 4 inter-operator threads and from 1 to 32 intra-operator threads. We chose these configurations because the average width of the LSTM network is of 2,7 operators per layer. According to the guidelines of the original paper [5], the point of best performance for a NN with such width would be at 3 inter-operator threads and around 21,3 intra-operators per thread (64 physical CPUS / 3 pools).

From that experiment, we found out that the point of best performance was at 2 inter-operator threads and 25 intra-operator threads. These results closely resembled the ones from the reference paper, and contained the most important information to our interest: performance decreases after that point, when more threads are added. In other words, there was oversubscription, the effect that happens when synchronization times are high compared with execution times. That was, precisely, what our program intended to fix.

Another curious part about these initial results was that we were seeing an improvement in performance every time either an intra or an inter operator thread was added, while the other threading configuration was set to 1. As an example, performances were improving every time an inter-operator thread was added while the intra-operator threads were set to 1. We wanted to know when these improvements in performance reached a halt, so we devised two experiments. The first experiment tested the performance of the use case when the intra-operator threads were set to 1 and the inter-operator threads ranged from 1 to 128 (the number of logical CPUs). From that experiment, the best result was achieved at the point with 8 inter-operator threads and 1 intra-operator thread. Then, the second experiment intended to obtain the same kind of information from the intra-operator settings. It included settings with 1 inter-operator thread and from 1 to 128 intra-operator threads. In that test, the point of best performance was reached at 51 intra-operator threads. By using this information, we were able to narrow down the number of settings we were going to use for testing.

An aspect that was very important for the fidelity of our results was to avoid using extra CPUs in unnecessary situations. For example, in a setting with 6 inter-operator and 4 intra-operator threads, a maximum of 24 CPUs can be used at the same time. In that situation, if more than 24 CPUs are assigned to the task, the extra CPUs will be used to run other processes of the operating system, leading to reduced runtimes. This shortening, however, won't be representative of the actual runtime it would've taken with just 24 CPUs, it will be less. Hence, we created specific CPU assignations to every threading configuration, so that each task used up to the maximum number of possible CPUs for that setting. In addition, we also disabled busy waiting, which occurs when a CPU is in an intermediate state between active and inactive. In that state, the CPU is constantly checking if it can or not continue a process started by another thread, and it does so by means of a loop. In our use case, and with our program, this is unnecessary and could consume more resources than needed. Therefore, we disabled this feature. As previously mentioned, all the code can be found annotated in the GitHub[1] repository.

After finishing all this code refinement, we ran four different experiments: The first one was the LSTM use case without any parallelism, the second enabled parallelism in one section, the third enabled parallelism in the two main computing-intensive sections, and in the foruth we applied our parallelism program to the third. All these experiments are explained in detail in the *results and discussion* section.

4. Evaluating an alternative solution

As far as we are concerned, there is only one existing alternative solution to improve the performance of parallelism features in DL frameworks, and it also requires a malleable implementation of intra and inter-operator thread settings. This alternative solution is presented in a paper called *Towards a malleable Tensorflow implementation* [20]. The principal idea behind this work is very closely related to the one used in our work: the fact that the selection of intra and inter-parallelism features is fixed in all DL frameworks hinders their flexibility and performance. Then, they explain what the necessary modifications would be to enable a dynamic selection of threads in Tensorflow. The main difference between their work and ours is that we are doing a malleable implementation of the framework at a Linux kernel level, instead of changing the code of the libraries in charge of the framework's parallelism. In other words, we are approaching the same idea but from a fundamental level. This level is the Linux kernel, which is in charge of creating, eliminating and assigning threads. By approaching malleability from that level, we are able to improve over many limitations of their work, which they present in their conclusions and future work section. From the six limitations listed in their work, we have been able to resolve four, only by working at a kernel level. Another limitation we've managed to solve relates to applying these improvements to a real-world scenario, which is precisely what we've done in this work by using an LSTM use case. However, we've not yet applied our program to GPUs, one limitation listed in their work that we've yet to tackle. The four limitations we've been able to fix by just working at a kernel level are:

- To provide an integration with a co-scheduler:

A co-scheduler is a higher-level resource manager that assigns resources dynamically. It doesn't really make sense to do a dynamic implementation of a DL framework without using a co-scheduler. From the library-level implementation of malleability, the co-scheduler will have to be added separately to the DL framework. In contrast, our kernel-level approach already includes the co-scheduler, improving performance and programmability.

- Creation of a malleability API:

In the mentioned paper [20], they believe that an API should be created with the objective of selecting the intra and inter-operator settings at any stage of the NN training and inference. Nonetheless, we believe that this is not necessary because our program automatically selects the best threading configuration at each step. Moreover, contrary to their work, everything we do in terms of threading and parallelism at a kernel level can be printed and visualized very easily with a simple command. This command is present in the GitHub[1] repository. On top of that, our program has the format of a library, which is what we have deemed more suitable to the application. A library is more convenient because our malleability add-on works at a kernel level.

- Management Through containers:

Once again, we've been able to manage all computing resources efficiently without re-programming an extra tool such as containers. If we would've worked at a library-level, we would've needed to create a malleable implementation of containers to the DL framework. By avoiding this unnecessary extra step in our work, we've improved the programmability, applicability and overall performance of our program.

- Intra-task malleability:

One of the main advantages of our approach is that by working at a kernel level, we can provide a dynamic management of both intra and inter-operator threads. If we had worked at a library level, we would have only been able to do a dynamic assignation of inter-operator

threads, which reduces by a lot the improvement in performance. This is one of the main constrains of the alternative solution, and we have been able to solve it without re-programming an intra-operator threading library.

All in all, we've seen how the dynamic implementation of parallelism features in DL frameworks can be done at different levels, which provides a potential alternative solution to our work. However, the best performing, most effective and programmable solution is to apply malleability at a kernel level. By working at that stage, the allocation of computing resources is instantly managed for upper levels, without having to do any other specific programming. Hence, we've chosen this method of work, which we want to continue improving upon.

5. Results and discussion

In this section, we will present the results of our investigation together with a detailed explanation. As previously mentioned, we have performed four principal experiments with four different levels of parallelism. These experiments have been divided into four subsections:

5.1 LSTM use case without threading

The first experiment consists of the runtime results of the LSTM use case without any threads. In other words, these results reflect the runtime of the LSTM script without any parallelism configurations explicitly enabled. Because of that, the program runs linearly, using two CPUs at most: one that's running the use case and another one running operations from the OS.

Below, you can find the results of the program's runtime, together with the statistical analysis of the results.

Experiment	Run nº1	Run nº2	Run nº3	Run nº4
Time(s)	35,19	32,67	33,88	33,05

Table 4 – Runtime results of the LSTM use case without any threads enabled. The first run is considered as a warmup run, used to warm up the computer for the experiments.

Before performing a statistical analysis on the results presented in table 1, we will introduce the different statistical functions and terms used.

- Mean time (s): $\mu = \frac{\sum x}{N}$ Where N is the number of samples and $\sum x$ is the sum of the value of each sample x.
- Standard Deviation (s): $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$ Where x_i corresponds to each sample's value and the other variables remain the same.
- Coefficient of Variation: $CV = \frac{\sigma}{\mu}$ Where all variables remain the same.
- Margin of error (s): $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{N}}$ Where all variables remain the same.

Mean time (s)	Standard Deviation (s)	Coefficient of Variation	Margin of Error (s)
$\mu = 33{,}2$	$\sigma = 0{,}51$	CV = 0,02	$\sigma_{\bar{x}} = 0{,}29$

Table 5 – Statistical analysis of the runtime results from table 1. The analysis contains information about the mean time (s), standard deviation (s), coefficient of variation and margin of error (s).

From the statistical analysis of the results, we gathered one of the most crucial pieces of evidence: that there are minimal runtime differences between executions of the same use case. On the one hand, in computer science a standard deviation of only 0,51 seconds over a 30 second runtime is considered as ideal, considering the nature of the use case and the hardware platforms used [4][8][9][12]. On the other hand, we also have a very low margin of error and coefficient of variation, reassuring our confidence in these results. It must also be mentioned that we didn't use the first run, since it was the warmup run and its information is not as reliable since the system is setting up for execution. Therefore, the conclusion is

that we have a reliable method to extract and reproduce results, both in terms of hardware and scripting.

We will later see how this configuration is the worst performing one, and why. All the code used in this section is present in the GitHub[1] repository of this study, under *First_experiment* section.

5.2 The LSTM use case with a forked section

In the *CPU threading and torchscript inference* [15] tutorial from the Pytorch team, the concept of a fork is introduced for threading and parallelism. Briefly, a fork can be thought as a ramification of a program's workflow. Adding forks to a program can enable parallelism at different levels. In this second experiment, a fork is added to parallelize the forward and backward layers of the LSTM module. That way, each iteration of the network, for both training and inference, can have one of its workflow's sections parallelized. Below, we present the runtime results of this program, together with the corresponding statistical analysis.

Experiment	Run nº1	Run nº2	Run nº3	Run nº4
Time(s)	84,80	83,88	85,09	84,74

Table 3 – Runtime results of the LSTM use case with a forked section. The first run is considered as a warmup run, used to warm up the computer used for the experiments.

Mean time (s)	Standard Deviation (s)	Coefficient of Variation	Margin of Error (s)
$\mu = 84{,}59$	$\sigma = 0{,}51$	CV = 0,01	$\sigma_{\bar{x}} = 0{,}29$

Table 4 – Statistical analysis of the runtime results from table 3. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s).

Following the same trend as in the previous experiment, the statistical analysis of this results proves the reproducibility of our experiments, with an even lower coefficient of variation. In terms of the results themselves, we can see how the fork has doubled the runtime of the use case. That is because each iteration of the workflow's forked part must go through an extra function. Since we haven't enabled the multi-threading settings yet, the fork increases the runtime of the use case. That is because the program can only do a linear run and use up to two CPU's, one for the computer processes and another one for the use case itself. In the next experiments, we'll see how adding more threads resolves that problem. Once again, the full code of this experiment can be found in the GitHub[1] repository of this work under the *Second_experiment* section.

5.3 The LSTM use case with two forked sections and threading settings

In this experiment, another fork is added to a different section of the program, called the LSTMEnsemble. As the name indicates, the LSTMEnsemble oversees an ensemble of predictions, with the objective of outputting the best result. This section is yet another one that can be parallelized in the LSTM use case, hence the addition of a fork. However, there are no more sections that could be performed in parallel. In this part, we will present the runtime results for that experiment with and without threading settings enabled. Firstly, we will present the runtime results without thread settings, in the same format as we presented them in the two previous sections.

Experiment	Run nº1	Run nº2	Run nº3	Run nº4
Time(s)	152,05	154,73	154,01	153,89

Table 5 – Runtime results of the LSTM use case with two forks and no thread settings. The first run is considered as a warmup run, a run that is used to warm up the computer used for the experiments.

Mean time (s)	Standard Deviation (s)	Coefficient of Variation	Margin of Error (s)
μ = 154,21	σ = 0,37	CV = 0,002	$\sigma_{\bar{x}}$ = 0,21

Table 6 – Statistical analysis of the runtime results from table 5. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s).

As we can see, these results follow the same trend as the previous one: adding a fork to a parallelizable section increases the execution time. In parallel, we obtained even better results from the statistical analysis, with improvements in the three key parameters for reproducibility. Now, we will present the runtime results with added thread settings. The details and justification for all the configurations we've used in the following experiment can be found in section 3.2.3 of the methodology part of this study. The two thread settings we added corresponded to intra and inter-operator parallelism, and used the functions detailed in the official Pytorch threading tutorial [15]. The results are in a tabular format.

	16	33,09	116,05	66,28	40,54	28,31	36,14	52,92	92,71	145,96
	8	34,59	115,54	65,65	40,39	27,96	34,41	51,43	92,31	156,21
	4	61,54	115,26	64,84	38,78	28,1	31,47	43,84	69,21	126,95
	2	120,94	113,4	66,57	42,18	28,09	25,61	28,7	43,76	81,95
Inter	1	238,72	124,52	72,22	47,29	29,64	26,85	27,53	26,96	32,77
		1	2	4	8	16	24	32	48	64
		Intra								

Table 7 – Runtime results for the LSTM use case with two forked sections using different thread settings. The X axis corresponds to intra-operator settings, from 1 to 64 intra-operator threads. The Y axis corresponds to inter-operator settings, from 1 to 16 inter-operator pools or threads. The green cell is the best execution runtime.

The results presented in table 7 correspond to the execution runtime of the LSTM use case with two forked sections and threading settings enabled. Each one of the results is the average of three runs, without counting the additional warmup run. There's no statistical analysis of these results because reproducibility had already been established with the first experiment.

There are two key insights to be drawn from the results of table 7, which are:

1. That by obtaining these values, we've managed to reproduce most of the initial paper our work is based upon. In Figure 2 of this work, which belongs to that paper [5], the performance trends match almost exactly what can see in table 7. For comparison, we've created table 8, which is a representation of table 7 normalized to the cell with one intra and one inter-operator. In that table, we can see how increasing the number threads leads to better performance, up until the point of best performance, which is at twenty-four intra-operator threads and two inter-operator pools. Excluding the first column, the distribution of performance resembles a Gaussian distribution.

2. That oversubscription is present, hence there's room for improvement by our dynamic threading program. After the point of best performance, all configurations with more threads present worsened results. The reason behind this effect is oversubscription, a consequence of adding more threads that's extensively discussed in our work.

	16	7,21	2,06	3,60	5,89	8,43	6,61	4,51	2,57	1,64
	8	6,90	2,07	3,64	5,91	8,54	6,94	4,64	2,59	1,53
	4	3,88	2,07	3,68	6,16	8,50	7,59	5,45	3,45	1,88
	2	1,97	2,11	3,59	5,66	8,50	**9,32**	8,32	5,46	2,91
Inter	1	1,00	1,92	3,31	5,05	8,05	8,89	8,67	8,85	7,28
		1	2	4	8	16	24	32	48	64
		Intra								

Table 8 – This table displays the normalization of the results from table 7 to that of the cell with one inter-operator pool and one intra-operator thread (238,72 seconds). The highlighted value is the point of best performance.

5.4 The LSTM use case with two forked sections, threading settings and our program enabled.

Finally, we arrive to the last experiment, which is the same as the third one but with our program for dynamic threading enabled. First, we performed a statistical analysis of the results to ensure reproducibility. The runs from table 9 correspond to the setting of one intra-operator thread and one inter-operator thread. We've executed these runs and the following statistical analysis to ensure reproducibility.

Experiment	Run nº1	Run nº2	Run nº3	Run nº4
Time(s)	235,49	236,07	236,19	235,65

Table 9 – Runtime results of the LSTM use case with a forked section. The first run is considered as a warmup run, used to warm up the computer used for the experiments.

Mean time (s)	Standard Deviation (s)	Coefficient of Variation	Margin of Error (s)
μ = 235,97	σ = 0,23	CV = 0,001	$\sigma_{\bar{x}}$ = 0,13

Table 10 – Statistical analysis of the runtime results from table 9. The analysis contains information about the mean time (s), standard deviation(s), coefficient of variation and Margin of error(s).

Once more, the statistical analysis of the three runtimes ensures the reproducibility of the experiment. This time, the results are the best from the four experiments, with the lowest standard deviation.

The complete results of the fourth experiment are shown in table 11. In the table, we have included runtime values of execution for the LSTM use case with two forked sections and both the threading settings and our program enabled. In the same way as the last section, we haven't included any statistical analysis because the first experiment already shows reproducibility. Below, you can find the results in table 11 and the normalized version in table 12.

	16	35,04	120,98	71,39	53,16	37,23	27,17	26,28	23,82	32,91
	8	34,63	128,77	76,89	50,97	36,31	25,61	25,6	23,08	32,21
	4	62,61	116,62	67,68	48,65	34,39	26,09	25,57	22,97	31,13
	2	123,03	114,6	70,99	43,86	29,39	24,26	26,25	24,28	31,86
Inter	1	235,97	127,47	75,36	50,06	35,16	32,18	33,28	34,42	43,18
		1	2	4	8	16	24	32	48	64
		Intra								

Table 11 - Runtime results for the LSTM use case with two forked sections using different thread settings and enabling our custom dynamic threading program. The X axis corresponds to intra-operator settings, from 1 to 64 intra-operator threads. The Y axis corresponds to inter-operator settings, from 1 to 16 inter-operator pools or threads. The green cell corresponds to the execution time of the setting with best performance.

	16	6,99	2,02	3,43	4,61	6,58	9,02	9,32	10,28	7,44
	8	7,07	1,90	3,19	4,81	6,75	9,57	9,57	10,61	7,61
	4	3,91	2,10	3,62	5,04	7,12	9,39	9,58	**10,66**	7,87
	2	1,99	2,14	3,45	5,59	8,33	10,10	9,33	10,09	7,69
Inter	1	1,00	1,92	3,25	4,89	6,97	7,61	7,36	7,12	5,67
		1	2	4	8	16	24	32	48	64
		Intra								

Table 12 - This table displays the normalization of the results from table 9 to that of the cell with one inter-operator pool and one intra-operator thread (244,96 seconds). The highlighted value is the point of best performance.

If we go back to table 7, we can see how, without our program, the best performing setting achieves an average execution time of 25,61 seconds. Then, in table 11, which presents the same experiment as table 7 but with our program enabled, we can see that the best time is of 22,97 seconds. Below, in table 13, we've made a comparison table to present the performance difference of these point in the two experiments. With these results, we have achieved the objective of our work: to improve the performance of the Pytorch 1.13.1 DL framework.

	Experiment nº 3	Experiment nº 4	Percentage difference
Best execution runtime (s)	25,61	22,97	10,31 %
Standard Deviation (s)	σ = 0,37	σ = 23	-
Maximum difference (s)	25,98	22,74	12,47%
Minimum difference (s)	25,24	23,20	8,08%

Table 13 – This table shows the percentage difference between the best execution runtimes of experiments nº3 and nº4. The maximum and minimum differences correspond to the highest and lowest possible runtime differences respectively, taking into account the standard deviation of each experiment.

Table 13 displays these results as a percentage. In those terms, our dynamic threading program has improved the best performance of Pytorch 1.13.0 by an astounding 10,31%. If we take into account the standard deviation of each experiment, our program achieves a minimum improvement of an 8,07% and a maximum of 12,47%. In following sections, we will discuss the implications of this big step in performance.

Then, in table 14 we wanted to display which are the configurations from the third experiment that perform better than table 11's configuration with the most threads. The idea behind this experiment is to show that, with our program, selecting the setting with the highest number of threads is a better solution than most configurations that run without base Pytorch 1.13.1. The implications for that will be discussed in more detail in the following sections.

	16	7,21	2,06	3,60	5,89	8,43	6,61	4,51	2,57	1,64
	8	6,90	2,07	3,64	5,91	8,54	6,94	4,64	2,59	1,53
	4	3,88	2,07	3,68	6,16	8,50	7,59	5,45	3,45	1,88
	2	1,97	2,11	3,59	5,66	8,50	9,32	8,32	5,46	2,91
Inter	1	1,00	1,92	3,31	5,05	8,05	8,89	8,67	8,85	7,28
		1	2	4	8	16	24	32	48	64
		Intra								

Table 14 – This table showcases the normalized results from table 8 with those performances higher than 7,44 highlighted in green. The selection of 7,44 corresponds to the performance of the configuration with the most threads in the fourth experiment.

As we can see in table 14, most configurations from the third experiment perform worse than the configuration with most threads from the last experiment. More precisely, the configuration with the most threads with our program enabled performs better than 77,77% of the executions without our program. That is, 35 out of 45 runs from the third experiment perform worse. As we will see, this has profound implications.

6. Budget summary and/or economic feasibility study

This study has been done in collaboration with the BSC, where I worked for a year as a Master Student researcher. My company provided all the computing infrastructure and the technical expertise to carry out the experiments. Therefore, there have been no other costs to the project, other than what the BSC has used in terms of resources, which is unavailable to me as an employee. However, I have been able to make some approximate estimations of what would've been the total costs of my investigation, which are present in table 15. These estimations have been made after a literature review, in combination with the AWS prices [21][22][23][24].

	Compute Time	Compute cost (128 CPU)	Worker time	Worker cost	Infrastructure cost
1 Year / Personal	400	1,5€	960 hr.	7,5 €/hr.	4000€ / yr.

Table 16 – Yearly personals costs working at the BSC.

According to the table, in one year I've consumed approximately 11800€ worth of resources, the total cost of this investigation. It must also be mentioned that the BSC is a research center funded by Univesitat Politència de Catalunya, the Spanish government, and the Catalan Government.

In terms of the economic feasibility of the study, it would be very impractical to reproduce it without the infrastructure of a supercomputer and the technical expertise to perform experiments on such a platform. However, provided all of that, this study could be reproduced using the code present in the GitHub[1] repository.

7. Analysis and assessment of environmental and social implications

Before jumping into the consequences of our work, it is necessary to reiterate what we have achieved. Firstly, we have improved the performance of the Pytorch 1.13.1 DL library by at least an 8%. Secondly, we have found that maxing out the number of threads when our program is enabled provides a performance superior to 77,77% of configurations without our software. That way, if performance is not paramount, the user will know he can achieve competitive performance without having to scan through all the configurations. Because of these two results from our work, our dynamic threading program fully satisfies the two types of Pytorch users: those who need to fully improve performance, and those who don't require or have the time to do it. And, most importantly, it does so automatically by just adding and activating a library to the desired code.

In the following subsections, we will address the different economic, environmental, and social implications of our work, together with the necessary documentation to support it.

7.1 Economic implications

In economic terms, our results can translate into huge cost reductions both for companies and individual users. To put it into perspective, we've created two different use cases where our program can reduce costs.

7.1.1 Training a model with a supercomputer

The first use case is that of an organization training a large-scale model using a supercomputer. In that setting, around 1024 high-end CPUs would be used for a total duration of 34 days, with a total cost for training of approximately 5 million USD [21]. By reducing an 8% of the training time, the total training period would be shortened by three days, which would save 600000 USD.

7.1.2 A company using AWS

The second use case is that of a company using Amazon Web Services (AWS) for its ML applications. According to different sources [22][23][24], the average price per hour of two 128 CPU cluster is 11,34 USD. A medium-sized company uses about 250 hours of ML cloud computing every month, using at least two 128 CPU clusters [25][26]. Therefore, the monthly cost that cloud computing has to the company is 5670 USD. By using our program, we could reduce that time by a minimum of an 8%, lowering the usage to 230 hours a month. With that, the cost would go down to 5216,4 USD, saving a total of 453,6 USD a month. In addition, developers wanting to test their models could also make use of our program to quickly apply a competitive setting, saving even more time and money.

7.2 Environmental

From an ecological perspective, reducing training and inference times would translate into lowering the power consumption of running a DL program. According to multiple studies [27][28][29], in ML applications, the average power consumption of a CPU is approximately 170 Watt per hour. In terms of CO_2, other sources [30][31][32] indicate that, if all electricity generation sources are averaged, the mean CO_2 emissions per Watt/h would be 0,296 g. This means that a CPU running for an hour produces 50,32 g of CO_2, without counting the losses in transportation caused by the Joules effect. If we can reduce the usage of a CPU by at least an 8%, a DL program that would've taken an hour to run on a CPU, now it would run for only 55,2 minutes. Those 4,8 minutes of reduced runtime account for 4,03 g of CO_2. If we take the example from section 7.1.1, by shortening the execution time three days, we would reduce CO_2 emissions by 3,6 kg. By comparison, during those three days it would

take 18 trees to absorb such an amount of CO_2 [33]. If looked at scale, it is clear how our program can have a great impact in the efforts to reduce carbon emissions, which is one of the major concerns of supercomputing, artificial intelligence, business intelligence, and the machine learning field.

7.3 Social Implications

The possible social implications of our work are endless, because a development in technology always leads to a myriad of unexpected social changes. Nonetheless, there are some aspects of social life that would undoubtedly benefit from our program. As we've mentioned before in our work, one of the main challenges of DL is improving its performance on CPUs for its deployment on mobile and edge devices. A performance increase of at least and 8% means that many DL models and applications that can't yet run on these devices, now can. Some of the devices that could be taken as an example include smartphones, smartwatches, and self-driving cars. In the following subsections, we'll dive into these three different examples.

7.3.1 Smartphones

In the case of smartphones, there are some applications that can't yet be launched because mobile CPU processors are not able to handle them. These applications include vision recognition, speech recognition, image filtering, language translation and more. By including these applications in smartphones, the ways in which we interact can be radically changed.

7.3.2 Smartwatches

To smartwatches, an 8% improvement in the performance of DL models means that these devices could now train and infer on biometrical data with much more sophistication and accuracy. By doing that, better health metrics could be acquired, improving the physical condition of its users.

7.3.3 Self-driving cars

The last example would be self-driving cars, which are severely constrained by the processing power of their CPUs [34]. These types of cars need to make thousands of inference processes offline, and in certain situations they must prioritise certain inference tasks over others because their CPUs can't handle everything. With our improvements, self-driving cars would be able to process way more information, allowing for a safer ride to both the car users and everyone else. Our program could help achieving fully autonomous self-driving cars, which would radically change the world of transportation.

8. Conclusions

The main conclusion to be drawn from this study is that we've managed to improve the performance of Pytorch 1.13.1 by exploiting the parallelism opportunities of neural networks. On the one hand, we've demonstrated that our program can improve over the best Pytorch configuration by at least an 8%. On the other hand, we've proven that it also outperforms 78% of Pytorch's thread configurations when we assign it the maximum number of threads. Both feats satisfy all Pytorch users by delivering the best performance, the best usability, and the best programmability. To achieve that, we've developed a program capable of dynamically changing thread usage through a network's training and inference processes. In contrast, there are no major deep learning frameworks that allow the assignation of threads to a network in a dynamic way. The implications of our results affect all aspects of life, ranging from a reduction of a company's costs to making self-driving cars drive safer.

There are many aspects of our study that we would like to address in future work. To begin with, we would like to know what the effects of memory allocation to performance are. Several studies [17] have pointed out that allocating memory close to the CPU using it can improve execution runtimes, so we would like to add it to our project. Then, we would also like to better understand certain differences in performance that we don't yet fully comprehend. However, the next big step would be to test our software with many other different types of neural networks, DL use cases and configurations. By doing that, we would ensure the usability of our product over a wide variety of applications. Finally, we would really like to see our program being used in a HPC study, where we believe it can be of great use.

9. References

[1] Zhou, Zhi-Hua. Machine learning. Springer Nature, 2021.

[2] Shinde, Pramila P., and Seema Shah. "A review of machine learning and deep learning applications." 2018 Fourth international conference on computing communication control and automation (ICCUBEA). IEEE, 2018.

[3] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. 2018. Bandana: Using non-voltile memory for storing deep learning models. arXiv preprint arXiv:1811.05922 (2018).

[4] Arpan Jain, Ammar Ahmad Awan, Quentin Anthony, Hari Subramoni, and Dhableswar K. D. K. Panda. 2019. Performance characterization of DNN training using Tensorflow and PyTorch on modern clusters. In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'19). IEEE, 1–11.

[5] Wang, Yu Emma, et al. "Exploiting parallelism opportunities with deep learning frameworks." ACM Transactions on Architecture and Code Optimization (TACO) 18.1 (2020): 1-23.

[6] Wang, Yu, Gu-Yeon Wei, and David Brooks. "A systematic methodology for analysis of deep learning hardware and software platforms." Proceedings of Machine Learning and Systems 2 (2020): 30-43.

[7] Xi, Sam, et al. "SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads." ACM Transactions on Architecture and Code Optimization (TACO) 17.4 (2020): 1-26.

[8] Mittal, Sparsh, Poonam Rajput, and Sreenivas Subramoney. "A survey of deep learning on CPUs: opportunities and co-optimizations." IEEE Transactions on Neural Networks and Learning Systems (2021).

[9] Farrell, Steven, et al. "MLPerf™ HPC: A Holistic Benchmark Suite for Scientific Machine Learning on HPC Systems." 2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC). IEEE, 2021.

[10] Kim, Young Geun, and Carole-Jean Wu. "Autofl: Enabling heterogeneity-aware energy efficient federated learning." MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. 2021.

[11] Li, Shen, et al. "Pytorch distributed: Experiences on accelerating data parallel training." arXiv preprint arXiv:2006.15704 (2020).

[12] Jain, Arpan, et al. "Performance characterization of dnn training using tensorflow and pytorch on modern clusters." 2019 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2019.

[13] Hewett, Russell J., and Thomas J. Grady II. "A linear algebraic approach to model parallelism in deep learning." arXiv preprint arXiv:2006.03108 (2020).

[14] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).

[15] CPU threading and TorchScript inference. PyTorch 1.13 documentation. (n.d.). Retrieved December 14, 2022, from https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html

[16] Dynamic parallelism in torchscript. Dynamic Parallelism in TorchScript - PyTorch Tutorials 1.13.0 documentation. (n.d.). Retrieved December 14, 2022, from https://pytorch.org/tutorials/advanced/torch-script-parallelism.html

[17] Jean Cho, M., & Saroufim, M. (n.d.). “Grokking pytorch Intel CPU performance from First principles”. PyTorch Tutorials 1.13.0 documentation. Retrieved December 14, 2022, from https://pytorch.org/tutorials/intermediate/torchserve_with_ipex.html

[18] Migacz, S. (n.d.). “*Performance tuning guide*”. PyTorch Tutorials 1.12.1 documentation. Retrieved December 14, 2022, from https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html

[19] Schütt, Thorsten, Florian Schintke, and Alexander Reinefeld. "Efficient synchronization of replicated data in distributed systems." International Conference on Computational Science. Springer, Berlin, Heidelberg, 2003.

[20] Libutti, Leandro Ariel, et al. "Towards a malleable tensorflow implementation." Conference on Cloud Computing, Big Data & Emerging Topics. Springer, Cham, 2020.

[21] Brown, Tom, et al. "Language models are few-shot learners." Advances in neural information processing systems 33 (2020): 1877-1901.

[22] Baughman, Matt, et al. "Deconstructing the 2017 changes to AWS spot market pricing." Proceedings of the 10th Workshop on Scientific Cloud Computing. 2019.

[23] George, Gareth, et al. "Analyzing AWS spot instance pricing." 2019 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2019.

[24] Amazon Web Services. (2006). ES. Amazon. Retrieved January 6, 2023, from https://aws.amazon.com/es/ec2/pricing/

[25] Rashid, Aaqib, and Amit Chaturvedi. "Cloud computing characteristics and services: a brief review." International Journal of Computer Sciences and Engineering 7.2 (2019): 421-426.

[26] Novais, Luciano, Juan Manuel Maqueira, and Ángel Ortiz-Bas. "A systematic literature review of cloud computing use in supply chain integration." Computers & Industrial Engineering 129 (2019): 296-314.

[27] Andrae, Anders SG. "Prediction studies of electricity use of global computing in 2030." International Journal of Science and Engineering Investigations 8.86 (2019): 27-33.

[28] García-Martín, Eva, et al. "Estimation of energy consumption in machine learning." Journal of Parallel and Distributed Computing 134 (2019): 75-88.

[29] Pramanik, Pijush Kanti Dutta, et al. "Power consumption analysis, measurement, management, and issues: A state-of-the-art review of smartphone battery and energy usage." IEEE Access 7 (2019): 182113-182172.

[30] Rodrigues, Joao FD, et al. "Drivers of CO2 emissions from electricity generation in the European Union 2000–2015." Renewable and Sustainable Energy Reviews 133 (2020): 110104.

[31] Malla, Sunil. "CO2 emissions from electricity generation in seven Asia-Pacific and North American countries: a decomposition analysis." Energy Policy 37.1 (2009): 1-9.

[32] Zhang, Ming, et al. "Decomposition analysis of CO2 emissions from electricity generation in China." Energy policy 52 (2013): 159-165.

[33] Li, Jian-Feng, et al. "CO2 absorption/emission and aerodynamic effects of trees on the concentrations in a street canyon in Guangzhou, China." Environmental pollution 177 (2013): 4-12.

[34] Farag, Wael, and Zakaria Saleh. "An advanced vehicle detection and tracking scheme for self-driving cars." 2nd Smart Cities Symposium (SCS 2019). IET, 2019.