Comparison between Ray and PyTorch: Python scaling frameworks for Big Data Analysis and Machine Learning

Aikaterini Liagka
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el17208@mail.ntua.gr

Apostolos S. Tzellas
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19878@mail.ntua.gr

Nikolaos Liagkas
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el19221@mail.ntua.gr

***Abstract*—This paper presents a comparative analysis of two Python scaling frameworks, Ray and PyTorch, in the context of big data processing and machine learning tasks. The study evaluates their performance across a series of distributed experiments, including k-means clustering, PageRank, and image classification on chest X-rays for pneumonia detection. Through these experiments, we assess the ease of use, memory management, time performance, and scalability of each framework. The results indicate that Ray excels in handling large datasets and distributed tasks with superior memory efficiency and automated cluster management. In contrast, PyTorch demonstrates better performance in complex deep learning tasks, benefiting from its optimized operations and GPU acceleration. These insights provide a comprehensive understanding of the strengths and weaknesses of each framework, guiding developers in selecting the most suitable tool for their specific needs in large-scale machine learning and data analysis applications.**

***Index Terms*—python, scaling frameworks, Ray, PyTorch, big data, machine learning, k-means, pagerank, image classification, pneumonia detection**

I. Introduction

In today's digital age the majority of people are interacting with the Internet daily for various reasons. From social media, online shopping and entertainment to financial transactions, work and research, it seems like the digital world has dominated every aspect of the modern everyday life. This has led to the explosive generation of massive loads of data from various sources, which created the urgent need for efficient ways to store, manage and analyze big data within a tolerable elapsed time and with sustainable, and often distributed, resources.

The availability of these extensive datasets, combined with the advances in computational power, are key contributing factors in the rapid advancement of machine learning these past few years. The continuous research has led to more sophisticated algorithms and models with impressive ability to learn patterns and make accurate predictions. As a result, machine learning is now widely used in numerous fields to solve previously intractable problems and enable significant technological breakthroughs.

These are some of the reasons why currently the vast majority of scientific research and technological innovation has been centered around machine learning and big data. Therefore several frameworks have been developed to handle and simplify these tasks. Python, as one of the most dominant languages in data science, offers many different frameworks for this purpose. It can be quite challenging for someone to choose the right framework in this wide range of options, with each of them having different strengths and unique features.

To address this challenge, in this paper we will perform a comparative analysis of two such Python scaling frameworks: Ray and PyTorch. In a distributed environment we will evaluate the performance of these frameworks across several experiments, using different data types and sizes, on various number of nodes. Our experiments include popular machine learning and graph operators, like clustering and PageRank. We also conducted a more complex experiment involving image classification on X-Ray images to detect pneumonia. It should be clear that, even though we made a conscious effort to make our experiments as realistic as possible and provide meaningful results, our main goal is not to assess the accuracy of the algorithms, but evaluate the performance of the two frameworks in terms of scaling. Therefore the main purpose of our research is to highlight the strengths and weaknesses, as well as the differences of Ray and PyTorch, providing insight into the optimal use cases for each framework in real-world scenarios.

II. Code Availability

The scripts we utilized for the experiments of this paper can be found in our GitHub repository [1], available at *https://github.com/LiagkaAikaterini/Ray_VS_PyTorch*. This repository contains all the necessary code for replicating our experiments comparing Ray and PyTorch frameworks.

The directory structure is organized by framework, making it easy to navigate and locate specific scripts. Particularly the two main directories are *Ray* and *PyTorch*, each consisting of 3 sub-directories named kmeans, pagerank and pneumonia_classification. Those are dedicated to the experiments we conducted and include the python scripts that we used, as well as the *res* directory where the txt files of our experiments' outputs can be found.

In addition, our GitHub repository contains two more directories named *data* and *documents*. The code for the resize of our data, which is explained in the Datasets section, is located in the data directory, whereas the documents directory includes the project assignment and our report.

III. Infrastructure and Software Overview

A. Okeanos-Knossos [2]

Okeanos-Knossos is a cloud service for the Greek Research and Academic Community, provided by the GRNET. It delivers production-quality IaaS, with virtual infrastructures and is offered to the Greek academic community for free. Okeanos-knossos enables users to deploy custom virtual machines and networks, manage storage, create diverse environments and network topologies without the need for permanent hardware. It provides a powerful, flexible, and user-friendly environment, making it easy to explore different technologies and distributed systems.

B. Apache Hadoop [3], [4]

Hadoop is a framework for scalable and distributed processing of large datasets. It offers a highly available service across a cluster of computers, while simultaneously detecting and handling failures. In this research we leveraged a module that Apache Hadoop offers, called Hadoop Distributed File System (HDFS). HDFS is a distributed file system with a master/slave architecture, which provides reliable storage, as well as high throughput access to data, and is suitable for applications that have large datasets.

C. Python

Python is one of the most prominent languages in data science and machine learning applications. It is versatile and easy to use, featuring a low learning curve and, most importantly, extensive library support. Python provides a big variety of powerful libraries and frameworks for data analysis and manipulation, machine and deep learning operations, big data processing, data visualization, NLP, automation and many other tasks. The main focus of our paper is to compare two of these Python frameworks, Ray and PyTorch, but within the scope of our research we used other Python libraries as well, some of which are scikit-learn, numpy, pandas and pyarrow.

D. Ray [5]

Ray is an open-source unified framework designed to scale AI and Python applications, like machine learning. It minimizes the complexity of managing distributed systems, as it provides a compute layer for parallel processing across multiple cores and nodes. Moreover, Ray encapsulates built-in libraries for data processing, model training, hyperparameter tuning and reinforcement learning, that facilitates efficient scaling of machine learning models. With its user-friendly API and the automated handling of key processes, such as scheduling, it simplifies scaling across large clusters. Ray also offers many advantages for working efficiently with datasets, for instance, the Ray Datasets component provides high-performance data processing API for large-scale datasets.

E. PyTorch [6]

PyTorch is an open-source framework, widely popular for its dynamic graph computations and deep learning capabilities using both GPUs and CPUs. The tensor library is one of PyTorch's core features. Tensors are multi-dimensional arrays, similar to NumPy arrays, which also provide GPU acceleration, storage efficiency and easy manipulation of data for complex tasks. PyTorch has a wide variety of other libraries for machine learning tasks, such as computer vision, natural language processing (NLP), neural network training etc. Additionally, it provides the torch.distributed backend, which enables scalable distributed training and performance optimization across multiple machines.

IV. Installation and Setup

A. Virtual Machines

The first step to conduct our experiments, which involve parallel data processing, is to build a distributed environment. As NTUA students we were able to utilize okeanos-knossos resources, which were assigned to our team after we logged in with our academic credentials and joined the course's project. We created 3 virtual machines with:

- Ubuntu Server 16.04 LTS
- 4 CPUS
- 8 GB RAM
- 30 GB Disk size

B. Network Configuration

We connected all of the machines to the same private network and one of them was assigned a public IP address. The machine with the public IP will be the master of our cluster and will act as a router to provide internet connection to the other two slave machines. To accomplish that, we followed the corresponding okeanos-knossos user guide [7], starting by identifying these specific configuration for our system:

- eth1 is the private network card.
- eth2 is the public network card.
- 192.168.0.1 is the private IP of the master machine
- 192.168.0.2:22 is the private IP (192.168.0.2) and port (22) of the slave1 machine we want to connect to.

- 192.168.0.4:22 is the private IP (192.168.0.4) and port (22) of the slave2 machine we want to connect to.
- we chose 10022 and 13389 as the ports of the router VM that will forward requests to slave1 and slave2 respectively.

These configurations will of course differ in other systems. They can be identified using the command:

```
lshw -class network
```

To enable NAT and connect through the router VM to a specific port of a NATed machine we executed the commands below, only in the master machine.

```
sudo bash -c 'echo "1" > /proc/sys/net/ipv4/ip_forward'
sudo bash -c 'iptables -F'
sudo bash -c 'iptables -t nat -F'
sudo bash -c 'iptables -t nat -A POSTROUTING -o eth2 -j
     MASQUERADE'
sudo bash -c 'iptables -t nat -A POSTROUTING -o eth1 -j
     MASQUERADE'
sudo bash -c 'iptables -t nat -A PREROUTING -p tcp --dport
     10022 -j DNAT --to-destination 192.168.0.2:22'
sudo bash -c 'iptables -t nat -A PREROUTING -p tcp --dport
     13389 -j DNAT --to-destination 192.168.0.4:22'
```

Then in each of the slave machines we add the master as the default gateway, by executing the command:

```
sudo bash -c 'route add default gw 192.168.0.1'
```

C. Ubuntu

The operating system image available in okeanos-knossos was Ubuntu Server 16.04 LTS, an older version of Ubuntu that comes with Python 3.5. This posed issues regarding the installation of Ray, as the framework required Python 3.9 or newer versions. To overcome this compatibility issue and also be able to utilize all the available features and optimizations in newer software versions, we decided to update the Ubuntu to its newest version. This was accomplished by sequential system upgrades, from version 16.04 to 18.04, then to 20.04 and lastly to the version used for the current paper's experiments, which is Ubuntu 22.04.4 LTS. The commands used to upgrade the operating system version are presented in order below.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get dist-upgrade
sudo apt-get autoremove
sudo apt-get install update-manager-core
sudo do-release-upgrade
```

D. Cluster Setup

In a distributed environment it is crucial for the machines to have the ability to communicate autonomously, without the need of a password, in order to access resources and perform necessary tasks for the execution of the experiments.

Initially we change the hostname of each virtual machine with the command :

```
sudo hostnamectl set-hostname <custom_hostname>
```

We then configure the private IPs with the new respective hostnames in each machine's */etc/hosts* file. For our 3-node cluster system the following lines must be added:

```
192.168.0.1 master
192.168.0.2 slave1
192.168.0.4 slave2
```

After saving the changes in all the machines, in order to apply the new network settings, we execute:

```
sudo systemctl restart networking
```

Lastly, with the subsequent commands, we will configure SSH-key based authentication between the machines. After locally generating an SSH key pair in the master, the public key is distributed to the slave machines, thereby enabling passwordless login from the master to both slaves.

```
ssh-keygen
ssh-copy-id user@slave1
ssh-copy-id user@slave2
```

This process is replicated to each slave, with the necessary command adjustments, to ensure uninterrupted communication between all nodes during the experiments.

E. Additional Dependencies

Beyond Ray and PyTorch, the installation of the following dependencies is necessary to support the execution of our experiments.

```
sudo apt install python3-pip
pip3 install scikit-learn
pip3 install torch_ppr
pip3 install numpy
pip3 install pyarrow
pip3 install pillow
```

The Installation of the Java Development Kit (JDK) was a prerequisite for setting up Hadoop.

```
sudo apt install default-jdk
```

After all the dependencies are installed we update and upgrade the system to get the latest software versions and ensure the system's security and stability.

```
sudo apt-get update
sudo apt-get upgrade
```

F. Hadoop

The installation of Apache Hadoop requires numerous configurations. Therefore in this section we will describe the steps we took for its setup. It is important to note that this process should be applied on all the machines within the system.

Initially we downloaded all the compressed files from the official Apache Hadoop website, extracted its contents and placed them in a new directory named hadoop for simplicity.

```
wget http://apache.cs.utah.edu/hadoop/common/current
    /hadoop-3.4.0.tar.gz

tar -xzvf hadoop-3.4.0.tar.gz
mv hadoop-3.4.0 hadoop
```

We then have to add the environment variables to set up Java and Hadoop paths. This involves identifying the path to the java installation, which can be achieved with the command:

```
readlink -f /usr/bin/java | sed "s:bin/java::"
```

We proceed to modify several files by adding the appropriate configurations, in order to specify environment settings and ensure proper integration of Hadoop within our system. The specific files, as well as their required modifications are listed below. However, anyone attempting to install Hadoop to a different system should adjust the paths, as well as the private IP address of the master machine accordingly.

- Inside the directory *hadoop/etc/hadoop* we add to the files *hadoop-env.sh*, *mapred-env.sh* and *yarn-env.sh* the following line:

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

- *hadoop/etc/hadoop/core-site.xml* :

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://192.168.0.1:50000</value>
    </property>
</configuration>
```

- *hadoop/etc/hadoop/hdfs-site.xml* :

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/home/user/hadoop/data/nameNode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/home/user/hadoop/data/dataNode</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

- *hadoop/etc/hadoop/yarn-site.xml* :

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.shuffle.
             class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler
             </value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>192.168.0.1</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>192.168.0.1:8032</value>
    </property>
</configuration>
```

- *hadoop/etc/hadoop/mapred-site.xml* :

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

- *.bashrc* :

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
export HADOOP_HOME=/home/user/hadoop
export PATH=${PATH}:${HADOOP_HOME}/bin:${HADOOP_HOME}/
     sbin
```

- *.profile* :

```
PATH=/home/user/hadoop/bin:/home/user/hadoop/sbin:$PATH
```

  To reload the configuration settings and paths immediately in the current session we execute

```
source .bashrc
. .bashrc

source .profile
. .profile
```

- Only in the master machine the file *hadoop/etc/hadoop/-workers* should contain only the hostnames of the nodes in our cluster, as shown below:

```
master
slave1
slave2
```

Finally, the Apache Hadoop, HDFS and Apache Yarn can be launched across the cluster by executing the following commands exclusively in the master machine.

```
hadoop/bin/hadoop namenode -format
hadoop/sbin/start-all.sh
```

After the successful installation and initialization of Apache Hadoop, we stored all the data files we leveraged for our experiments to HDFS.

Firstly, within HDFS we created the directory */data/*, where all the dataset files will be located:

```
hdfs dfs -mkdir /data
```

After uploading the data files in the master machine's local disk storage, we loaded them into the directory */data/* in HDFS using this command adjusted appropriately :

```
hdfs dfs -put /path/to/local/files /data/
```

G. Ray Setup

We installed the official release of Ray, with all its components, to gain access to all the libraries offered by the framework. That entails the default components Core, Dashboard and Cluster Launcher along with Ray Data, Train, Tune, Serve and RLlib. In every machine we executed the command :

```
pip3 install -U "ray[all]"
```

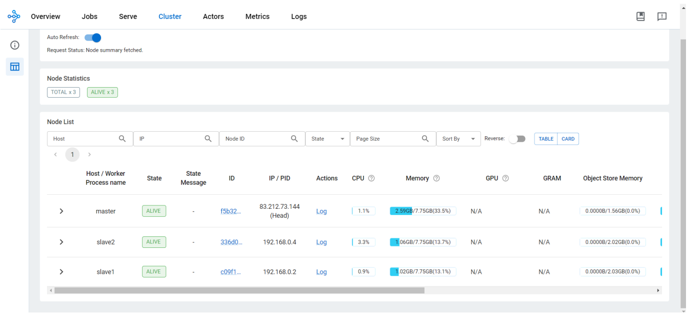
After the installation is complete, the Ray Cluster must be launched. The head of the cluster, which in our case is our master machine with private IP 192.168.0.1, is explicitly defined by running the command :

```
ray start --head --port=6379 --dashboard-host=
    0.0.0.0 --dashboard-port=8265
```

Then both our slave machines are added to the cluster with the following command, where the head node's IP and the communication port are defined :

```
ray start --address=192.168.0.1:6379
```

In the head's command, in addition to the start of the Ray Cluster, the Ray Dashboard's web interface is configured, by defining its host address and port number. Ray Dashboard provides a visual interface that displays the real-time system metrics. The user can monitor the resources that are utilized in each node, the job profiling and the tasks of their Ray application.



Request Status: Node summary fetched.

Node Statistics

TOTAL x 3 ALIVE x 3

Node List

	Host / Worker Process name	State	State Message	ID	IP / PID	Actions	CPU	Memory	GPU	GRAM	Object Store Memory
>	master	ALIVE	-	f5b32...	83.212.73.144 (Head)	Log	1.1%	2.59GB/7.75GB(33.5%)	N/A	N/A	0.0000B/1.56GB(0.0%)
>	slave2	ALIVE	-	336d0...	192.168.0.4	Log	3.3%	1.06GB/7.75GB(13.7%)	N/A	N/A	0.0000B/2.02GB(0.0%)
>	slave1	ALIVE	-	c09f1...	192.168.0.2	Log	0.9%	1.02GB/7.75GB(13.1%)	N/A	N/A	0.0000B/2.03GB(0.0%)

Fig. 1. Screenshot of Ray Dashboard's Interface

H. PyTorch Setup

Since the virtual machines in our cluster lack GPU resources, we installed the CPU-only versions of PyTorch, as it is specified in the official website, with the following command:

```
pip3 install torch torchvision torchaudio --index-url
    https://download.pytorch.org/whl/cpu
```

V. Datasets

To conduct our experiments, we utilized 2 real-world datasets with substantially large sizes and different data types. The first dataset consists of an undirected graph, whereas the second contains x-ray images. All the data files used in our experiments are stored in HDFS, inside the directory */data/*.

A. Friendster Social Network Dataset [8]

Friendster is an online gaming network, that launched as a social networking site where users could form friendship edges with each other. It also allowed the formation of groups, which members could create and join. This dataset, provided by Stanford University's SNAP, is 30.1 GB with 65,608,366 nodes and 1,806,067,135 edges. Each line of the data file contains 2 node identifiers, that represent an edge in this undirected graph.

In the scope of this study and with the resources we had access to, such large volume of data could not be handled, but we leveraged this real-world dataset to create smaller data files of various sizes. After downloading the *com-friendster.ungraph.txt.gz*, we resized it using the python script *data/resize.py* that can be found in our GitHub repository [1]. Executing this script will produce 4 new files, which will contain millions of records:

- *test_data.csv* : a 10 MB file that we used to test and verify our code's accuracy quickly. This file was not part of our experiments and its results will not be included in this paper.
- *data_1.csv* : 1.06 GB file, which consists of 66,781,840 edges and 11,191,238 individual nodes.
- *data_2.csv* : 2.65 GB file, with 163,532,542 edges and 21,148,573 individual nodes.
- *data.csv* : 10.5 GB file, including 620,186,769 edges and 41,480,685 individual nodes.

These undirected graph datasets will enable our team to conduct experiments with different dataset sizes and make observations on how that affects the performance. Our virtual machines are equipped with 8 GB RAM, which means that the 10.5 GB dataset clearly exceeds the main memory capacity. Although, considering the necessary RAM required for code execution, most of the times not even the 2.65 GB dataset could fit into main memory in its entirety.

B. Chest X-Ray Images (Pneumonia) [9]

This 1.26 GB Dataset was sourced from Kaggle and consists of 5,863 chest X-ray images (JPEG) classified in 2 categories, Pneumonia and Normal. It is organized into 3 folders, train, test and val, each of which contain subfolders for the 2 different image categories.

The X-ray images were selected from retrospective cohorts of pediatric patients, ages 1 to 5 years old, from Guangzhou Women and Children's Medical Center. All chest X-ray imaging was performed during the routine clinical care of the patients.

VI. Experiments

In this section we will examine the theoretical background and implementation details of our experiments' code. The main focus is to provide insight on the methods and algorithms we utilized, as well as the various programming decisions made during the development of the experiments. For unanswered questions or further code examination someone could also check the experiments' source code in our GitHub repository [1], which contains detailed comments.

A. Code Execution

Initially we will outline the fundamental structure and commands required, in order for a script to run successfully in a distributed environment with PyTorch or Ray.

- *Ray :*
  To connect to the existing cluster, in the beginning of each script Ray must be initialized using:

  ```
  ray.init(address='auto')
  ```

  In the end of each script, in order to properly terminate the Ray session and release the cluster resources, we use:

```
ray.shutdown()
```

To conduct an experiment using Ray, the script can be placed in the master machine. Then the developer will just execute the python script only on the master and Ray will automatically manage the distribution of code to all the active nodes of the Ray cluster. For example, a script located in the folder *Ray/kmeans* named *kmeans.py* can be executed with the command :

```
python3 Ray/kmeans/kmeans.py
```

If we want to run the experiment with fewer nodes, we will run the command mentioned above after removing one of the slave nodes from the cluster with:

```
ray stop
```

The node can reenter the cluster as it is indicated in the "Ray" section of "Installation and Setup".

- *PyTorch :*
  Using PyTorch's distributed package, every node must set up the distributed environment within every script. This is accomplished with the function below, where the world_size defines the number of nodes in the cluster and rank represents the unique identifier of each node. The identifier *"gloo"* is used to define that our backend will utilize CPUs, as there are no GPUs available in our resources:

```
def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = '192.168.0.1'
    os.environ['MASTER_PORT'] = '12345'

    # initialize the process group - gloo == cpu
    dist.init_process_group("gloo", rank=rank,
        world_size=world_size)
```

The variables *world_size* and *rank* are defined externally in the execution command and can be retrieved using the code below in the beginning of the script's main function.

```
rank = int(os.getenv('RANK'))
world_size = int(os.getenv('WORLD_SIZE'))
```

In the end of each script the following function is used to terminate the process group and properly shut down the distributed environment in PyTorch.

```
def cleanup():
    dist.destroy_process_group()
```

Performing an experiment using PyTorch in a distributed environment can be more challenging. We made sure to place the code in all the machines of the cluster. The execution command should be adjusted accordingly so that the argument *–nnodes* represents the total number of nodes in our cluster and the *–node_rank* is an integer from 0 to (nnodes - 1), acting as the unique identifier of each machine.
For example if the script is located in the folder *PyTorch/kmeans* named *kmeans.py* in all the machines, then to execute the script with 3 nodes we run:
On *"master"*, we make sure it is attributed *rank = 0* :

```
torchrun --nnodes=3 --node_rank=0 --master_addr='
    192.168.0.1' --master_port='12345' PyTorch/kmeans/
    kmeans.py
```

On *"slave1"*:

```
torchrun --nnodes=3 --node_rank=1 --master_addr='
    192.168.0.1' --master_port='12345' PyTorch/kmeans/
    kmeans.py
```

On *"slave2"*:

```
torchrun --nnodes=3 --node_rank=2 --master_addr='
    192.168.0.1' --master_port='12345' PyTorch/kmeans/
    kmeans.py
```

To perform the experiment with fewer nodes, the arguments need to be adjusted and the commands will be executed only on those specific nodes.

B. *K-Means Clustering*

In machine learning, clustering is used to group data in distinct groups based on their similarities. One of the most popular clustering algorithms is k-means, which partitions the dataset into k clusters. It iteratively assigns data points to the nearest cluster centroid, which is the mean of the data points assigned to that cluster, and then reevaluates the centroid's value. The k centroids are initially randomly selected and the process stops, either when their value does not change significantly between iterations, or when a specific number of iterations is reached.

In this experiment we used the 3 files created by the Friendster dataset, which indeed has user-defined groups [8]. The datasets were too large to be loaded into memory, so we employed the *pyarrow* library to read data from HDFS in predefined sized batches.

These batches were transformed into the right format and processed in parallel within our cluster, taking advantage of the *scikit-learn* library's modules. We utilized *KMeans*, a class in the sklearn.cluster module that implements the k-means algorithm and the function *calinski_harabasz_score* from sklearn.metrics to evaluate the clustering.

The Calinski-Harabasz Index is a useful metric that assesses the clustering results relying solely on them and the dataset, without the need for external, ground-truth labels. The between-cluster and within-cluster dispersion are determined, as high quality clusters should be compact and well separated from each other. In addition to the meaningful results it delivers, the Calinski-Harabasz Index has relatively low computational complexity, thus it was preferred over other metrics we tested, like Inertia or the Silhouette Score.

Our script performs the clustering in each batch individually and then evaluates it with the Calinski-Harabasz score. These scores, which separately offer an insight on local data patterns within each batch, are eventually averaged to provide a more global view of the clustering quality across the entire dataset.

The results of these experiments include the average Calinski-Harabasz score and the elapsed execution time, from the system initialization until the results are calculated. We also recorded the system initialization time for Ray and

Pytorch in each k-means clustering experiment, but the results were mostly random and that is why they will not be included in the analysis. Ultimately we call a custom *display_results* function, that prints and saves the results in a custom txt file.

The k-means script for PyTorch is located in the file *PyTorch/kmeans/kmeans.py*, while the Ray script is in *Ray/kmeans/kmeans.py* in our GitHub repository [1]. The discrepancies in the scripts are detailed below:

1) PyTorch: In the PyTorch script we read data from the csv file stored in HDFS, in 50 MB batches, which is a manageable size considering our resources. We also took advantage of the *Dataset*, *DataLoader* and *DistributedSampler*, offered by the torch.utils.data module, to handle the distribution of the data.

We recursively create a custom PyTorch Dataset for every 50MB batch with the *class GraphEdgeDataset(Dataset)* and with the use of a *DistributedSampler* we make sure every machine in our distributed setup will process a unique subset of this dataset. Then we load the *GraphEdgeDataset* and the *DistributedSampler* in a Dataloader with batch size 1024*1024 samples. Therefore the 50MB dataset was further partitioned into smaller chunks, which were equally distributed in our machines to be processed.

The data processing produces a list containing all the Calinski-Harabasz scores, from which a local average Calinski-Harabasz score is calculated in each machine. The final step is to gather the local scores to obtain the final result. To achieve this we encapsulated all the local average scores into tensors and then leveraged PyTorch's *dist.all_gather* to aggregate these tensors in a list. Ultimately only the master machine, with *rank == 0*, calculates the total average Calinski-Harabasz score from the gathered tensors and displays the results.

2) Ray: In Ray the batch size is 20MB, smaller compared to PyTorch for several reasons. The primary reason is that in this script we did not use a Dataloader that further partitions the data, so the KMeans algorithm was implemented on the whole 20MB batch, thus it required more processing resources. In addition, one of the machines in a Ray cluster acts as the coordinator that handles the data distribution and the job scheduling, which also occupies a part of this machine's RAM. The aforementioned resulted in Out Of Memory errors when we attempted to increase the batch size.

We used the decorator *@ray.remote* above the function *ray_kmeans* that was used to process the data in our Ray script. This alerts Ray that this function can be executed asynchronously and in parallel across the Ray cluster. We use the line :

```
results = ray.get([ray_kmeans.remote(batch.to_pandas
    ().values, config) for batch in csv_reader])
```

which ensures parallel execution, as the batches will be distributed across our machines. The function *ray.get()* is designed to handle the retrieval of results from the distributed tasks, so we did not need to gather the results manually. We just calculated the total average Calinski-Harabasz score by applying *np.mean* to the results list and subsequently displayed the results.

C. PageRank

PageRank is a graph operator, originally developed by Google, to rank web pages in their search engine results. It can be utilized to rank the importance of a node within a graph depending on its structure. The PageRank algorithm calculates the ranks, by firstly assigning a random rank to each node and then iteratively redistributing the ranks based on the incoming edges from other nodes, considering their importance.

In this experiment we used the 3 graph datasets derived from the Friendster social network once again. Just as in the previous experiment, we read the data with predefined batch sizes from the HDFS storage using *pyarrow*.

To execute the PageRank algorithm, we utilized the pre-built implementation *page_rank* provided by the package *torch_ppr* [10]. This implementation requires 2 lists of nodes as input, where the corresponding elements of the lists represent an edge in the graph. The nodes' IDs must be consecutive increasing integers, starting from 0. Our data did not conform to the required input structure, so we developed the function *input_format*, in both Ray and PyTorch, to fix the input format before calling *page_rank*. This function returns the input lists expected from torch-ppr page_rank, as well as a dictionary that maps node IDs to their corresponding consecutive indexes. This dictionary will eventually be used to decode the PageRank results and retrieve the original node IDs of our graph.

After executing the *page_rank* function for each batch, the scores were stored in a dictionary, where the key is the original ID of the node and the corresponding value is this node's PageRank score. Our idea to obtain the global ranking of a node was to aggregate all the scores from the different batches that contained this specific node. So if the node already exists in our result dictionary, we add the new score to the existing one. Finally, after the processing finishes, we will normalize all the scores for accurate results. However we encountered several issues related to the memory constraints of our system.

PageRank is a memory intensive task that our 8GB RAM could not perform while storing the enormous dictionary of the already calculated scores. To resolve this issue, we resorted to periodically storing the result dictionaries as json files at fixed intervals, which we achieved with the creation of the function *save_intermediate_results*. The intervals set for the Ray and PyTorch implementations were different. The results dictionary is cleared after the intermediate results are saved and is even followed by a call to *gc.collect()* to ensure memory relief.

After the processing is completed and all the PageRank score dictionaries are saved in json files, we used a function named *load_intermediate_results* to retrieve and aggregate them. Our objective was to display the top 10 most important nodes of our entire graph and their corresponding scores. Due to memory constraints, the bigger datasets' results could not fit into memory even when there were no PageRank calculation tasks taking place. Considering we wanted to display only the top 10 scores, in our *load_intermediate_results* function we recursively aggregated a pair of json files and retaining only the top 10.000 ranking nodes in each aggregation step. This method ensured that memory usage was kept within our system's limits, while still allowing us to accurately identify the top 10 scores across the entire graph.

In our GitHub repository [1] the PyTorch PageRank script is located in *PyTorch/pagerank/pagerank.py*, while the Ray script is in *Ray/pagerank/pagerank.py*. The differences between the two implementations are presented below.

1) PyTorch: The batch size we used in the *pv.ReadOptions* to read the data from HDFS is 50MB.

Once again we used *torch.utils.data* and created the custom PyTorch Dataset *class GraphEdgeDataset(Dataset)* according to the requirements of this experiment. With a Distributed-Sampler and a DataLoader we distributed the data across the machines after further batching it to chunks of 1024*1024 samples. We applied *page_rank* on each of these smaller batches and aggregated the results as it is described above.

After processing ten 50MB batches, the function *save_intermediate_results* was called for memory relief. We also made sure to save the last few batches even if the 10 batch interval was not reached. Once the processing is completed and all the intermediate files are saved, we call the function *load_intermediate_results*.

The last step is to gather the results from every machine in our distributed system. This is a little more complex than the k-means implementation, because in this experiment we have a dictionary of up to 10.000 nodes and scores. Knowing that a dictionary cannot be wrapped inside a tensor, we converted it into two lists. One list contains the nodes and the other contains the top scores, in corresponding positions. After encapsulating these lists into tensors with the correct dtypes, we firstly calculate their lengths. This step is necessary, because in PyTorch only tensors of the same size can be gathered in a distributed system. Therefore, we distribute all the tensor lengths between the machines using *dist.all_gather* and we pad all the tensors with dummy values up to the maximum length. Finally we proceed to gather all the results with *dist.all_gather*.

Only the master machine, with *rank == 0* will handle and display the gathered results. Firstly the padding is discarded and the results are aggregated to one dictionary. The top 10.000 ranking nodes and their PageRank scores are retained and as a final step we normalize the scores.

Finally, the top 10 scores with their corresponding node IDs are identified and displayed along with the execution time. This is handled by the function *display_results*, which prints and saves the results in a custom txt file in the directory *PyTorch/pagerank/res*.

It should be noted that in this script our *cleanup* function contains logic to delete all the intermediate result files generated during the execution, therefore ensuring efficient disk storage management.

2) Ray: In the Ray script, for the aforementioned reasons, we used a smaller batch size of 10MB. We tried to minimize the use of torch tensors and we did not utilize a Dataset, a DataLoader or a DistributedSampler, because the data is distributed automatically in the Ray cluster. We created the function *pagerank* and added the decorator *@ray.remote*, where the data formatting and the execution of the *page_rank* algorithm are performed asynchronously, in parallel.

Iterating through the batches we read from HDFS, we keep track of the tasks we submit for execution on a list named *futures* with the line :

```
futures.append(pagerank.remote(batch))
```

To avoid overwhelming the memory with numerous concurrent execution requests, which could result to an Out Of Memory Error, every 10 futures we execute the tasks with :

```
batch_results = [ray.get(f) for f in futures]
```

We aggregate the results in a dictionary and then empty the futures and batch_results lists. We also execute *gc.collect()* to make sure we relieve the memory.

We could save the intermediate results with our *save_intermediate_results* every 10 futures, but we decided to increase that interval to have a smaller I/O overhead. Consequently the results are iteratively saved in a json file every 30 batches and then the results dictionary is reset and *gc.collect()* is called. It has to be noted that we made sure the last batches are also saved, even if the 30 batch interval is not reached.

In the Ray framework the coordinator handles the retrieval of the results, which are automatically gathered in the master machine. Thus, the only thing we need to do is call the *load_intermediate_results* to successfully aggregate them and retrieve the top 10.000 ranking nodes, along with their PageRank scores. Finally, after we normalize the scores and keep the top 10 nodes, we call our *display_results* function to print and save the output in a custom txt file in the directory *Ray/pagerank/res*.

In the Ray script we also implemented a *cleanup* function, which only deletes all the intermediate json files created during the execution. This ensures that the disk storage is not consumed by redundant files.

D. X-Ray Image Classification

Lastly, we conducted a more complex experiment, discovered on Kaggle, which performs classification on X-Ray images with PyTorch, to identify pneumonia [11]. For this experiment we used the Chest X-Ray Images (Pneumonia) Dataset, which is 1.26 GB in size. In the previous tests, we evaluated various dataset sizes, so here we will use the entire dataset and observe the frameworks' performance with a different data type.

The original code on Kaggle is executed on a Notebook and it includes :

- *GPU acceleration logic*.
- The *data_transforms* function, which defines image preprocessing pipelines for training, validation, and testing phases. Resizing, cropping, rotation, normalization and conversion to tensors are a few of these modifications that are designed to enhance the model's performance.
- The class *classify(nn.Module)* defines a simple convolutional neural network (CNN) model that will be employed to classify chest X-ray images into two categories, pneumonia or not.
- The *Dataset loading* with PyTorch's data loader *datasets.ImageFolder*, that loads images from a directory where each subdirectory represents a different class. The appropriate transforms are applied and 3 datasets are created for different purposes, the trainset, testset and validset. Then the corresponding Dataloaders are created, with batch size 64 and the shuffling enabled.
- The *training loop*, where iterations over multiple epochs are performed using a specified loss function and optimizer. The losses of the training loop are defined by the *CrossEntropyLoss* function of the *torch.nn* module. This is a commonly used loss function that is mostly applied to determine how well or poorly a multi-class classification model is performing. The minimization of losses can be achieved by adjusting the model's parameters, task which is performed by the optimizer. In this experiment the optimizer is defined using the Adaptive Moment Estimation or *Adam*, from PyTorch's *torch.optim* module. "Adam" is an optimization algorithm that computes adaptive learning rates for each parameter. The learning rate is defined in the optimizer by the *lr* variable and in our experiments is set to *lr=0.01*. In the training loop the total loss for all the batches is accumulated and printed for evaluation purposes.
- The *testing* process involves iterating over the test images and making predictions based on the trained model. These predictions are compared to the true labels and the model accuracy is calculated as the ratio of the number of correctly classified images to the total number of images tested. Accuracy is a very important metric used to evaluate the training quality.

This was our baseline implementation and we kept many parts of the code intact, such as the data_transforms function and the classify(nn.Module) class. However we modified the original code to integrate it with the distributed architecture of our system. These modifications will be detailed, separately for each framework, below:

1) PyTorch: This script can be located in our GitHub repository [1] in the file *PyTorch/pneumonia_classification/pneumonia_classification.py*

The first modification we made was to remove all the GPU related logic, because our resources are not equipped with GPUs. We also added the necessary *setup* and *cleanup* functions, as they are described in the previous "Code Execution" section, to correctly establish the distributed environment.

Additionally, we will not use the datasets.ImageFolder, but once again we will utilize the *Dataset*, *DataLoader* and *DistributedSampler* from the torch.utils.data module, in order to correctly handle the data distribution in our cluster.

We created a custom PyTorch Dataset with the *class customDataset(Dataset)*, which emulates the behavior of the datasets.ImageFolder of the original code. The attributes of this class are:

- *self.hdfs* : An instance to "HadoopFileSystem" which is used to connect to HDFS, where the data are stored.
- *self.data_dir* : The exact location of the "Chest X-Ray Images" dataset inside HDFS.
- *self.transform* : The transform to be applied to each image as a preprocessing step during loading.
- *self.file_list* : A list containing the full file path of each image in our dataset.
- *self.classes* : A sorted list of all the class names, which represent the names of the subdirectories of our dataset.
- *self.class_to_idx* : A dictionary mapping each class name to a unique numerical label, which is a more suitable format for machine learning models.

The *self.hdfs*, *self.data_dir* and *self.transform* are passed as arguments externally, while the *self.file_list*, *self.classes* and *self.class_to_idx* are defined by a private method *_get_file_list_and_classes(self)*, which accesses the dataset within HDFS.

This customDataset implements the fundamental methods *__len__(self)* and *__getitem__(self, idx)*, which are mandatory for creating a *torch.utils.data.Dataset* class compatible with a *torch.utils.data.Dataloader*. The *__len__(self)* method calculates the length of the dataset by returning the length of the *self.file_list*. The *__getitem__(self, idx)* method retrieves a single data sample and its corresponding label by index, utilizing the attributes of our customDataset.

More specifically, the *__getitem__(self, idx)* method of a PyTorch Dataset is employed by the Dataloader to fetch individual samples for each batch. Initially our customDataset does not load all the images into memory, but stores their file paths instead, which is way less memory consuming. The Dataloader subsequently can use these file paths to load each individual image when the *__getitem__(self, idx)* method is called. This approach is preferable, because images will be loaded into memory in manageable batches. Our dataset is quite large, so loading its entirety in the beginning could compromise performance and even cause a memory overflow.

The 3 *customDatasets*, trainset, testset and validset, are created with the appropriate parameters. Then a *DistributedSampler* is created for each of these Datasets, to ensure the correct distribution of the data in all the machines within the cluster. The samplers will also handle the data shuffling, so that the data samples are fed into the model in a random order during each epoch, resulting in a robust training with the ability to generalize to unseen, random data and avoid overfitting. Lastly 3 *Dataloaders*, named trainloader, testloader and validloader, are created with the Datasets and their respective

DistributedSamplers. The batch size for these Dataloaders is set to 64 samples, equal to the batch size in the original script.

The model initialization is slightly different comparing to the original Kaggle code. After creating the model, we wrap it in *DistributedDataParallel* or DDP, which is provided by the *torch.nn.parallel* module of PyTorch :

```
model = classify().to(device)
model = DDP(model)
```

The DDP is a container that provides data parallelism by synchronizing gradients across each model replica, inside each machine of our cluster.

The training loop logic is put inside the *train* function, that returns the training loss for each epoch. We also add :

```
model.train()
```

as the first line of our train function to set the model in training mode and enable the appropriate features, such as Dropout and BatchNorm, which will improve the model's generalization ability and training quality. Before the train function is called, we add the line :

```
trainsampler.set_epoch(i+1)
```

where i is the current epoch. This ensures that each machine receives a new shuffle order for the data at the beginning of each epoch and is also synchronized with the other machines in our distributed environment, increasing the effectiveness of the training.

The train function will be executed on the data of the trainloader and the training loss for the epoch will be calculated. The next step is to gather all the losses from the different machines in our distributed system and aggregate them in one value, which was achieved with :

```
dist.all_reduce(gathered_running_loss, op=dist.
    ReduceOp.SUM)
```

Firstly we encapsulated the training loss in a tensor and then used the dist.all_reduce with the operation dist.ReduceOp.SUM. This results to all the machines having the same gathered_running_loss, which is the sum of the all the local training losses. Finally the gathered_running_loss is divided by the world size of the distributed environment, to calculate the average running loss across all the machines for this specific epoch. This result is added to the *result_text* that will be displayed and saved at the end of the execution.

After training the model the final step to evaluate the result is the testing. The testing logic of the original script can be located in the *test* function, which returns the number of the correct predictions, as well as the number of all the predictions, to calculate the accuracy. The only addition to the test logic is the line :

```
model.eval()
```

at the beginning of the function to secure consistent and reliable testing behaviour.

We gather the results of the test function with the same *dist.all_reduce* logic. Lastly the accuracy is calculated and added to the results_text, which only the master machine, with *rank == 0*, displays and saves with our custom *display_results* function.

2) Ray: The script is located in our GitHub repository [1] in the file *Ray/pneumonia_classification/pneumonia_classification.py*

In this experiment our Ray implementation incorporates numerous PyTorch components. PyTorch is a very powerful tool when it comes to deep learning functionalities and custom Dataset creation for better data management. Ray does not offer alternatives to fully replace the PyTorch elements without using a different framework similar to it, like TensorFlow, especially in a more complex deep learning experiment as the one we are conducting in this case. However our objective is to compare Ray with PyTorch, so involving a different framework while trying to remove PyTorch from the Ray script would be an inaccurate approach.

Therefore almost all the PyTorch components were kept intact and we added Ray elements, designed specifically for parallel PyTorch training, in order to observe how that would affect the performance. The main addition we made was the *TorchTrainer* [12] from the ray.train.torch module, which is a trainer that is supposed to simplify the PyTorch distributed training.

We created a *config* dictionary to specify all the arguments and hyperparameters in our training function, called *distributed_classification*. Another dictionary, called *scaling_config*, was also generated to configure how to scale data parallel training in our distributed environment. These dictionaries were passed as arguments to the TorchTrainer.

The *customDataset* was not altered, but we did not create *DistributedSamplers* as this is handled automatically by the TorchTrainer. We did create the 3 *Dataloaders*, as in the PyTorch script, adding the *shuffle = True* parameter to them, to make sure shuffling will be enabled.

To utilize the same code regardless of the number of workers in our cluster we added lines such as :

```
trainloader = ray.train.torch.prepare_data_loader(
    trainloader)

testloader = ray.train.torch.prepare_data_loader(
    testloader)
```

```
model = ray.train.torch.prepare_model(model)
```

These lines automatically prepare the model and Dataloaders for distributed execution if needed. That is the reason why we do not need to create the DistributedSamplers or wrap the model in DDP manually.

The sampler is now part of the Dataloader and is only created if there are more than 1 machines in the Ray cluster. Consequently, setting the epoch in the samplers during the training loop is slightly different :

```
if config['world_size'] > 1:
    trainloader.sampler.set_epoch(i+1)
```

We also used *ray.train.report* to report the metrics during the execution and the result_text, which we intended to save in a txt file at the end of execution.

It should be mentioned that we aggregated the results from the different machines using *dist.all_reduce*, exactly as it is described in the PyTorch script. While this step could probably be avoided, we wanted to ensure the accuracy of the final results, aligning them fully with the PyTorch implementation outcomes.

VII. Result Analysis and Evaluation

In this section, we will present the results of our experiments and conduct an in-depth analysis comparing Ray and PyTorch across multiple performance metrics.

A. K-Means Clustering

The result files of all the k-means clustering experiments can be found in our GitHub repository [1]. For the PyTorch experiments the files are located inside the directory *PyTorch/k-means/res*, whereas for Ray experiments in *Ray/kmeans/res*.

This experiment was conducted for each framework on all the 3 datasets derived from the Friendster Social Network Dataset, with sizes 1.06GB, 2.65GB and 10.05GB. For each dataset 3 experiments with different number of nodes were performed. Specifically the node number varied from 1 up to 3.

Consequently we run a total of 9 k-means clustering experiments for each framework. The metrics we will present to compare the performance of Ray and PyTorch are the execution time and the Calinski-Harabasz index.

The following graphs illustrate the correlation between the execution time and the number of nodes in the cluster. Each graph contains the results for a specific dataset size.

Analysis of figures 2, 3, 4

- **Time Performance:** Ray consistently performs better than PyTorch across all data sizes and number of nodes. The difference is significant as Ray is 10 to 13 times faster than PyTorch in every execution scenario. This could be attributed to Ray's very efficient memory management.
- **Number of Nodes:** Both Ray and PyTorch demonstrate significant time reductions when the number of active nodes are increased. That indicates that both frameworks can scale well with additional nodes to our distributed environment.

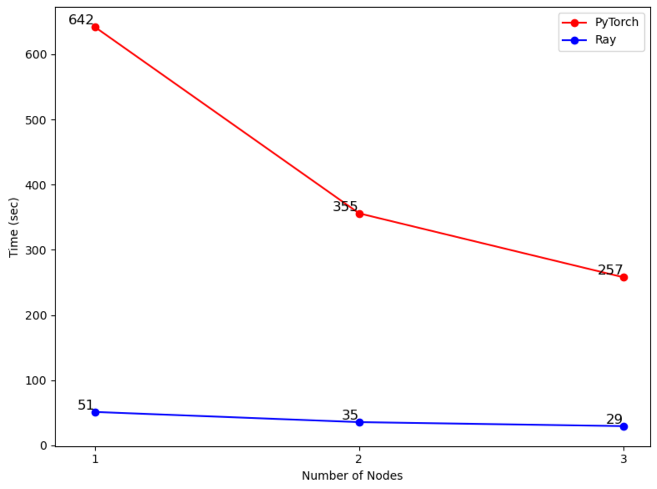


Fig. 2. Results for 1GB graph

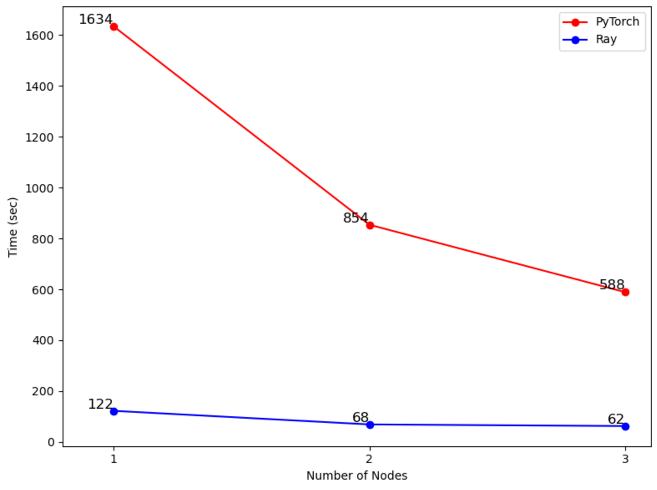


Fig. 3. Results for 2GB graph

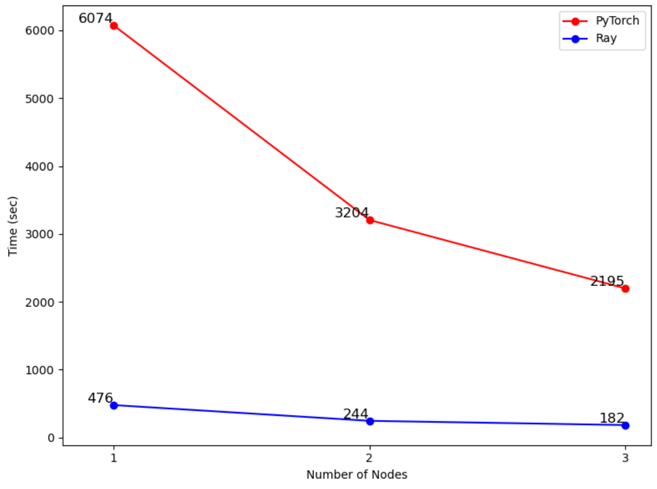


Fig. 4. Results for 10GB graph

We also plotted the subsequent graphs to depict the relationship between the execution time and the data size, for a fixed number of machines.

Analysis of figures 5, 6, 7

- **Data size:** Both Ray and PyTorch exhibit increased execution times for larger datasets. However, we can easily observe that PyTorch's execution time increases rapidly, amplifying the time performance disparity between the frameworks. This indicates that Ray manages data scaling better and is more suitable for larger datasets, when it comes to simple distributed machine learning tasks, such as clustering.

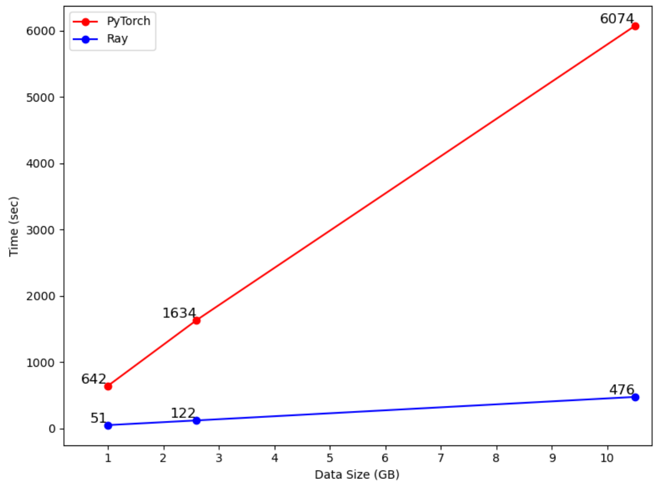


Fig. 5. Results for 1 Machine used

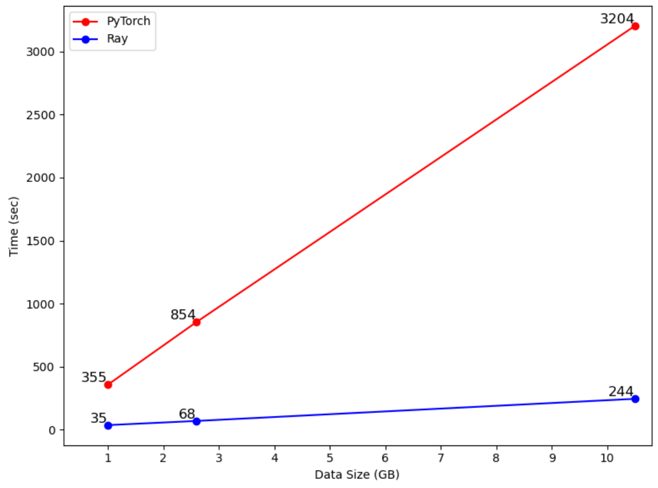


Fig. 6. Results for 2 Machines used

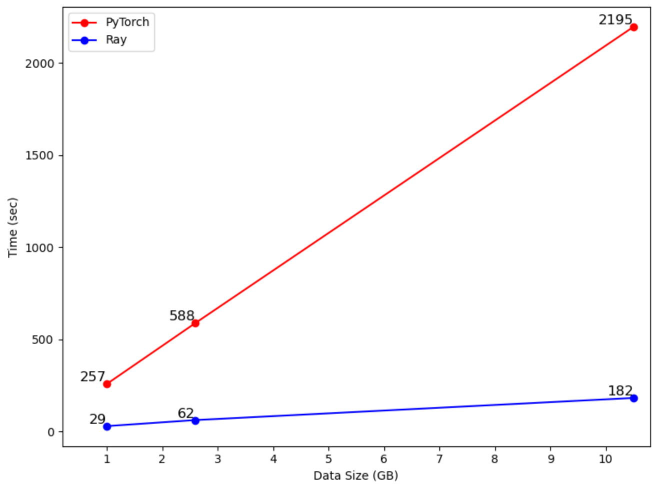


Fig. 7. Results for 3 Machines used

Analysis of Calinski-Harabasz Index

The Calinski-Harabasz scores for PyTorch and Ray are presented in the 2 following tables. It should be noted that higher scores reflect better quality clustering, as they indicate that the clusters are well-separated and compact.

We observe that no matter the number of nodes Ray has the same Calinski-Harabasz index for the same dataset. On the contrary the clustering quality varies in PyTorch, for different number of nodes in the same dataset. There is an obvious decline in quality from 1 to 2 nodes, which improves with the addition of the 3rd node. This suggests that PyTorch's distributed processing involves complexities that impact clustering performance, particularly in how data is distributed and processed across multiple nodes. A possible cause could be the way PyTorch's DistributedSampler distributes the data across the machines, resulting in slightly different batches for different number of nodes, which would affect the clustering quality.

Ray also exhibits higher Calinski-Harabasz scores than PyTorch for every experiment we conducted, highlighting that the k-means clustering quality is both more consistent and superior in Ray's implementation.

TABLE I
PyTorch Calinski-Harabasz Scores

	1 GB	*2,65 GB*	*10,5 GB*
1 Node	14,362,482	15,282,086	14,491,339
2 Nodes	11,473,656	11,711,595	10,943,274
3 Nodes	12,898,743	14,453,065	14,297,975

TABLE II
Ray Calinski-Harabasz Scores

	1 GB	*2,65 GB*	*10,5 GB*
1 Node	18,337,183	18,784,492	17,488,578
2 Nodes	18,337,183	18,784,492	17,488,578
3 Nodes	18,337,183	18,784,492	17,488,578

B. PageRank

For the PageRank experiments the result txt files are located in the directories *PyTorch/pagerank/res* for PyTorch and in *Ray/pagerank/res* for Ray.

Similarly to k-means, a total of 9 experiments were conducted for each framework, using 1, 2, and 3 nodes, across the same datasets with sizes 1.06GB, 2.65GB, and 10.05GB.

The result txt files also include the 10 most significant nodes, with the highest PageRank scores, for each experiment. We do not present these nodes in this paper, as we cannot extract valuable information for the frameworks' comparison from them.

The 3 following figures present the execution time in relation to the number of nodes, each for a different dataset size.

Analysis of figures 10, 11, 12

- **Time Performance:** Ray outperforms PyTorch again across all the differently configured experiments. Although their performance is closer than in k-means clustering, Ray is still approximately 4 times faster than PyTorch in every execution scenario.
- **Number of Nodes:** Both Ray and PyTorch decrease their execution time with the increase of active nodes. That indicates that both frameworks benefit from scaling, as nodes are added to our distributed environment.

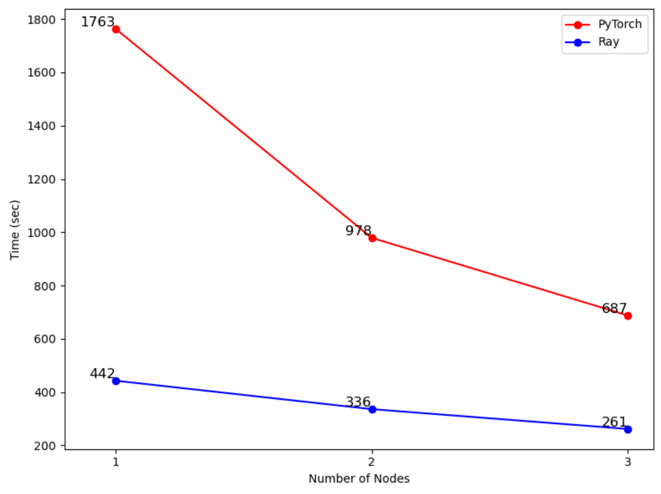


Fig. 8. Results for 1GB graph

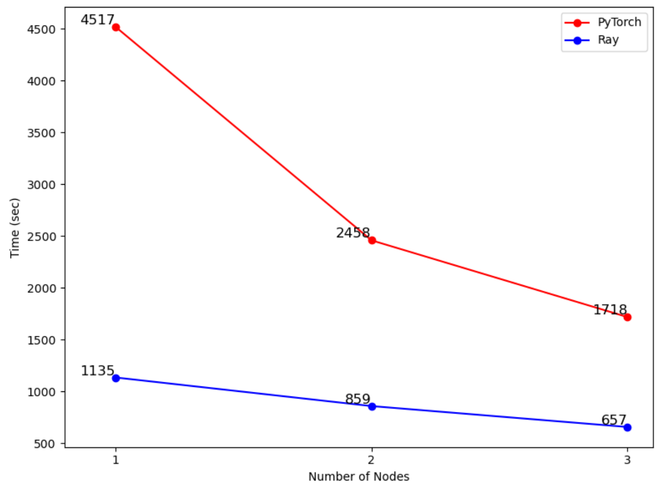


Fig. 9. Results for 2GB graph

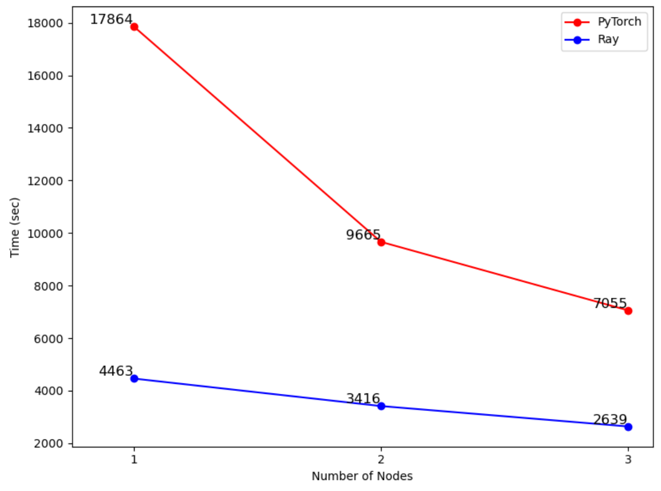


Fig. 10. Results for 10GB graph

Once again we also created plots to illustrate the execution time in relation to the number of nodes, for a fixed dataset size.

Analysis of figures 11, 12, 13

- **Data size:** As it was expected both frameworks exhibit increased execution time for larger datasets. However, as the dataset becomes larger the performance gap between PyTorch and Ray widens significantly. Thus, we conclude that data scaling is a lot better for Ray's implementation. In the PageRank experiments, which were more time consuming than k-means, Ray's superiority with larger dataset sizes was even more apparent. As it is shown on the graphs below, for the 10.5GB dataset, PyTorch's execution time ranged from 2 to 5 hours depending on the number of nodes, whereas Ray's execution time varied from 45 minutes to 1 hour and 15 minutes. This clearly indicates that Ray is the better choice for bigger datasets in simple distributed graph operator tasks, like PageRank.

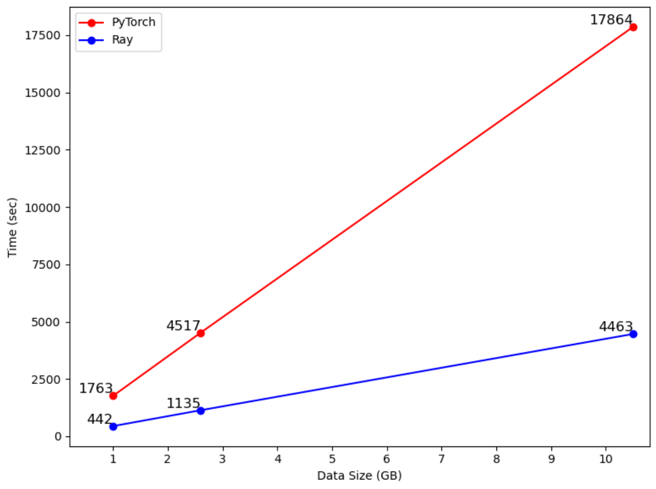


Fig. 11. Results for 1 Machine used

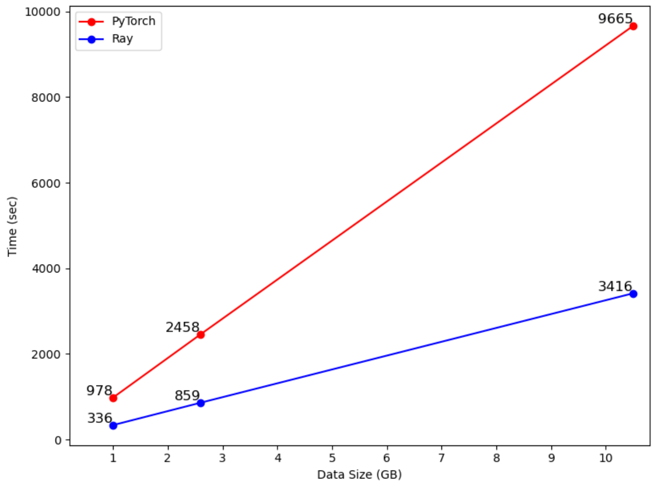


Fig. 12. Results for 2 Machines used

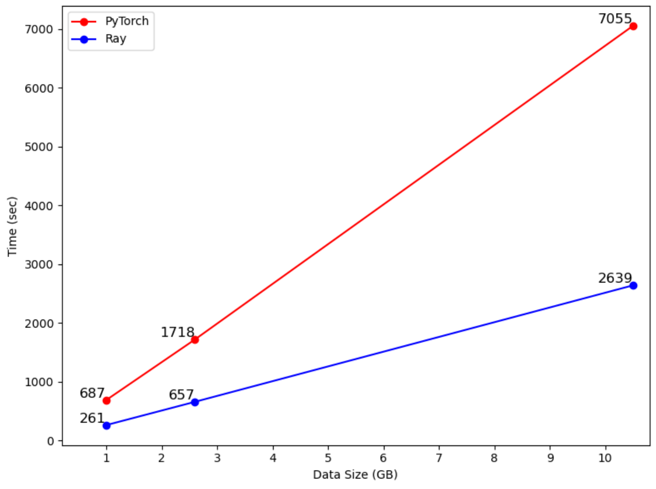


Fig. 13. Results for 3 Machines used

C. *X-Ray Image Classification*

The results for the X-Ray Image Classification experiments are located in our GitHub repository [1], inside the *PyTorch/pneumonia_classification/res* directory for PyTorch and in *Ray/pneumonia_classification/res* directory for Ray.

For the evaluation of the frameworks in this experiment we will compare the execution time, the losses during the training, as well as the accuracy of the model after testing.

Analysis of figure 14

- **Time Performance:** The following plot clearly shows that PyTorch has a better time performance than Ray in every one of these more complex classification experiments, no matter the configurations.
- **Number of Nodes:** Both frameworks scale well with the addition of nodes, as the performance time drops significantly in a similar way.

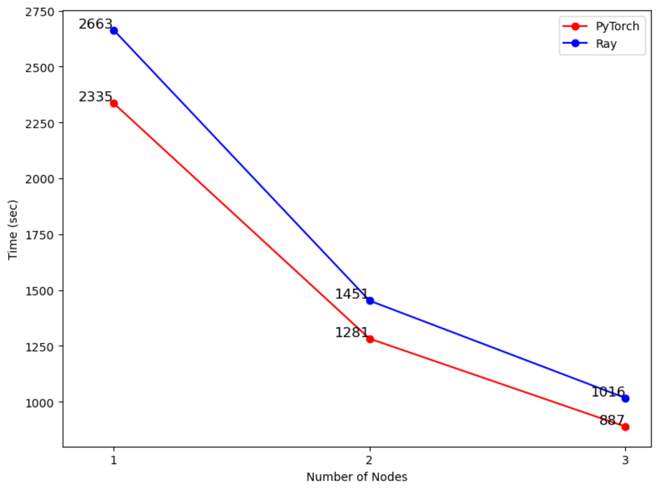


Fig. 14. Results for X-Ray Image Classification (Time)

Analysis of Training Losses

The training losses for each epoch are displayed in the following 2 tables, one for each framework. The valuable information we can extract is how fast the training losses converge to 0 while the epochs pass. The closer the training loss is to 0, the better the training quality of the model is, as long as there is no overfitting of training data, which does not happen in our experiments.

PyTorch's convergence seems faster for fewer nodes. This can be attributed to the framework's design, which focuses on efficient single-machine performance and GPU acceleration mainly for deep learning tasks.

On the contrary, when the distributed environment becomes larger, in our case for 3 nodes, Ray's training losses exhibit faster converge to 0. This highlights Ray's architecture, which is designed to effectively manage and optimize the execution of parallel and distributed workloads.

TABLE III
PyTorch Training Loss

	1 Node	*2 Nodes*	*3 Nodes*
Epoch 1	49.353	75.989	111.684
Epoch 2	6.189	6.973	19.890
Epoch 3	8.259	7.644	10.376
Epoch 4	3.792	6.751	6.172
Epoch 5	1.058	3.180	4.285
Epoch 6	0.381	2.807	4.887
Epoch 7	0.199	3.147	5.351
Epoch 8	0.169	2.860	4.230
Epoch 9	0.147	1.735	2.725
Epoch 10	0.142	1.587	1.745

TABLE IV
Ray Training Loss

	1 Node	*2 Nodes*	*3 Nodes*
Epoch 1	59.136	128.578	158.088
Epoch 2	12.698	12.047	14.457
Epoch 3	5.596	7.120	6.560
Epoch 4	2.494	5.446	4.130
Epoch 5	1.142	3.540	3.415
Epoch 6	0.460	6.825	3.063
Epoch 7	0.243	4.850	2.963
Epoch 8	0.193	3.240	4.282
Epoch 9	0.164	2.818	2.180
Epoch 10	0.131	2.190	1.459

Analysis of Accuracy

Specifically regarding the accuracy of the model, we will present results from 2 separate executions, as the initial results were considered insufficient to analyze and assess the frameworks' performance. This additional data enables us to conduct a more thorough and reliable analysis in the most important metric of this experiment.

It was deemed unnecessary to present the execution times and training losses of this second execution to this paper, as their patterns closely mirror the results we have already discussed, offering no additional insights.

We have to note that the variation in accuracy across different executions is justified, as the shuffling is enabled in the DistributedSampler and Dataloader, which introduces a controlled level of randomness in the creation of the data batches. This randomness is essential to achieve model generalization, thereby improving the overall training quality.

Someone can easily notice that although there is some randomness in the results, Ray achieves slightly higher accuracy in almost every execution scenario. The performance of the two frameworks is comparable, as the difference in accuracy is not big between them. We assume that the better results could be attributed to optimizations in communication or gradient synchronization in the Ray framework.

TABLE V
Model Accuracy Scores

	1st Run		**2nd Run**	
	PyTorch	***Ray***	***PyTorch***	***Ray***
1 Node	0.769	0.811	0.776	0.838
2 Nodes	0.792	0.766	0.780	0.812
3 Nodes	0.723	0.822	0.772	0.784

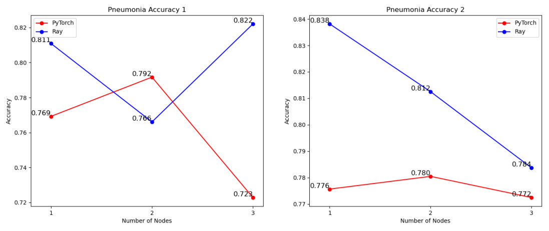


Fig. 15. Results for X-Ray Image Classification (Accuracy)

VIII. Comparison of Ray and PyTorch

In this section we will provide a comprehensive comparison, based on our experiments' result analysis and all the previously discussed aspects of the frameworks.

A. *Ease of Use*

PyTorch's **installation** process is very simple, whereas Ray requires the cluster setup which is slightly more complicated.

On the other hand, once the installation is complete, the **execution of a script** in a distributed environment using Ray is significantly easier. In PyTorch the user has to set up the environment with a function and also configure and pass all the environmental arguments in the *torchrun* command, while Ray handles all the configuration automatically.

In terms of ease of use, one of Ray's standout features is the **Ray Dashboard**. Ray's built-in web interface elevates the user experience by visualizing resource utilization and task distribution across the cluster. It can help developers monitor the execution of their scripts in real time and easily identify the cause of failures and bottlenecks.

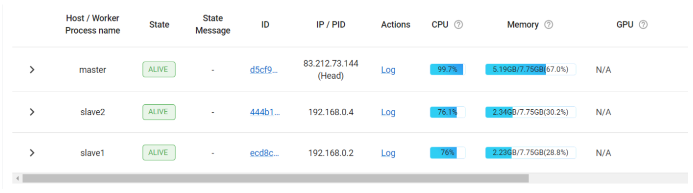
Generally, the two frameworks are designed to address different needs, which results in noticeably **different coding experiences** for the developer. PyTorch provides a lower level of abstraction, with greater control over the management of custom datasets and machine learning operations, but it requires developers to manually handle the computations and the data flow. On the contrary, Ray offers many automated, built-in operations to enable the easier management of large-scale, distributed workloads and machine learning tasks.

B. *Performance*

One of the most important metrics to evaluate a framework's performance is **memory efficiency**, especially when handling large datasets. Efficient memory management enables more complex models and larger datasets to be processed simultaneously without system crashes. A framework that optimizes memory usage not only enhances performance with the existing resources but also potentially reduces operational costs by minimizing the need for expensive hardware upgrades.

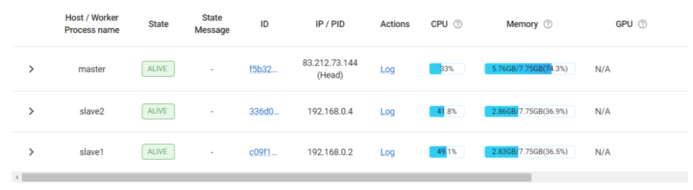
Ray Dashboard provides an interface that displays the memory usage dynamically, while a script is executed in the machines that participate in the Ray cluster. We were able to observe the memory usage both in the Ray and PyTorch script executions, since the Ray cluster was active throughout all our experiments.

The screenshots presented below, exhibit the memory usage during the execution of the PageRank experiment with 3 nodes, both in Ray and PyTorch, in the Ray Dashboard Interface.



	Host / Worker Process name	State	State Message	ID	IP / PID	Actions	CPU	Memory	GPU
>	master	ALIVE	-	d5cf9...	83.212.73.144 (Head)	Log	99.7%	5.19GB/7.75GB(67.0%)	N/A
>	slave2	ALIVE	-	444b1...	192.168.0.4	Log	76.1%	2.34GB/7.75GB(30.2%)	N/A
>	slave1	ALIVE	-	ecd8c...	192.168.0.2	Log	76%	2.23GB/7.75GB(28.8%)	N/A

Fig. 16. Ray Memory Usage



	Host / Worker Process name	State	State Message	ID	IP / PID	Actions	CPU	Memory	GPU
>	master	ALIVE	-	f5b32...	83.212.73.144 (Head)	Log	33%	5.76GB/7.75GB(74.3%)	N/A
>	slave2	ALIVE	-	336d0...	192.168.0.4	Log	41.8%	2.86GB/7.75GB(36.9%)	N/A
>	slave1	ALIVE	-	c09f1...	192.168.0.2	Log	49.1%	2.83GB/7.75GB(36.5%)	N/A

Fig. 17. PyTorch Memory Usage

It is apparent that the Ray framework manages memory more efficiently, as it utilizes 70%-95% of the CPU resources. On the contrary, the PyTorch framework is operating at 20%-45% of the available CPU capacity.

This disparity suggests that PyTorch may not be as efficient in distributing workloads or managing memory, potentially leading to underutilization of hardware and slower processing times in experiments run on CPUs.

However, it is important to note that PyTorch is specifically designed to leverage GPU acceleration. Therefore, if GPUs were available in our resources, PyTorch would probably demonstrate significantly higher memory efficiency and performance.

Another key metric in assessing a framework's performance, that was thoroughly examined in our results' analysis, is **time performance**, which refers to the efficiency and speed with which a framework executes tasks.

The best time performance between Ray and PyTorch varies depending on the experiment. Ray has significantly better time performances for simple machine learning and graph operating tasks, such as k-means clustering and PageRank. On the other hand PyTorch prevails in more complex deep learning experiments, like the X-ray image classification, which also required custom dataset handling.

Additional metrics we measured to evaluate our frameworks' performance were:

- the Calinski-Harabasz index for the clustering quality in the k-means experiments
- the training accuracy to assess the training quality of the model in the image classification experiments.

Generally Ray's metrics were higher than PyTorch's in almost every execution scenario. This indicates that Ray provides advantages in both clustering quality and classification performance. This may be a result of Ray's automated processes, which obviously contribute to more effective data handling and model training. In addition, Ray shows stability in the Calinski-Harabsz index, across different number of nodes, which suggests that it is more effective in maintaining clustering quality as resources are scaled up.

C. Scalability

Scalability refers to the ability of a system to efficiently handle increased workloads and expanded demands, without compromising the performance. We conducted experiments with different data types and sizes, with various number of nodes, to test the two frameworks' scalability.

By evaluating how the increase of data size affects performance, we were able to draw conclusions about **data scaling**, which is the ability to manage very large datasets. In every experiment it is evident that Ray handles data scaling more efficiently. As the datasets become larger, the time performance gap between the frameworks widens significantly, with Ray consistently outperforming PyTorch.

In addition, we will address **strong scaling**, which refers to the ability to reduce execution time as the data size remains the same and computational resources are added to the cluster. During our research, Ray and PyTorch exhibited similar strong scaling behaviour. Both frameworks benefited greatly from the addition of machines in the cluster and achieved much lower execution times.

Overall, **Ray is more scalable**, as it automatically handles distributed tasks. On the contrary, PyTorch's complicated execution process and manual handling requirements would make it really challenging to scale in large, complex clusters with many machines.

IX. Conclusion

Our research delved into an in-depth comparative analysis of Ray and PyTorch, highlighting each frameworks' strengths and weaknesses. We conducted various machine learning and graph related experiments, such as k-means clustering, PageRank and image classification, in an effort to evaluate and compare the two frameworks. This analysis aims to assist practitioners and researchers in making informed decisions when selecting a framework, that will best fit their needs in real-world scenarios.

The results of this paper revealed that Ray demonstrates ease of use, efficient memory management, automatic handling of distributed tasks and better scalability, which in most cases resulted in superior performance. However, PyTorch consistently outperformed Ray in the more complex deep learning experiments.

In conclusion, the choice between Ray or PyTorch ultimately depends on the specific Use Case. In large and complex distributed environments or for larger datasets Ray is the more suitable option, as opposed to PyTorch, which remains the robust choice for in-depth, GPU-accelerated machine learning tasks. It is important to emphasize that PyTorch offers certain deep learning and custom dataset management operations that Ray cannot replicate. Therefore, Ray integrates with PyTorch and includes functions to facilitate its use. Consequently, in many cases, both frameworks can be utilized simultaneously to leverage both their strengths, offering a complementary approach to complex machine learning tasks.

References

[1] A. Liagka, A. S. Tzellas, N. Liagkas, "Ray_VS_PyTorch," github.com. https://github.com/LiagkaAikaterini/Ray_VS_PyTorch.

[2] Okeanos-Knossos, Greek Research and Technology Network (GRNET), okeanos-knossos.grnet.gr. https://okeanos-knossos.grnet.gr/home/ (accessed: Aug. 24, 2024).

[3] Apache Software Foundation, "Apache Hadoop," apache.org. https://hadoop.apache.org/ (accessed: Aug. 24, 2024).

[4] D. Borthakur, "HDFS Architecture Guide," apache.org. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (accessed: Aug. 24, 2024).

[5] "Overview — Ray 2.34.0," docs.ray.io. https://docs.ray.io/en/latest/ray-overview/ (accessed: Aug. 24, 2024).

[6] PyTorch, pytorch.org. https://pytorch.org/ (accessed: Aug. 24, 2024).

[7] "How Can I Access All My VMs Using One Public IP (NAT)?," okeanos-knossos.grnet.gr. https://okeanos-knossos.grnet.gr/support/user-guide/cyclades-how-can-i-access-all-my-vms-using-one-public-ip-nat/ (accessed Aug. 25, 2024).

[8] J. Leskovec and A. Krevl, "Friendster social network and ground-truth communities," snap.stanford.edu. https://snap.stanford.edu/data/com-Friendster.html (accessed: Jul. 09, 2024).

[9] P. T. Mooney, "Chest X-Ray Images (Pneumonia)," kaggle.com. https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia/data (accessed: Aug. 17, 2024).

[10] M. Berrendorf, "torch-ppr," torch-ppr.readthedocs.io. https://torch-ppr.readthedocs.io/en/latest/ (accessed Jul. 29, 2024)

[11] F. Mehfooz, "Pneumonia Classification Using Pytorch," kaggle.com. https://www.kaggle.com/code/fahadmehfoooz/pneumonia-classification-using-pytorch (accessed Aug. 17, 2024).

[12] "ray.train.torch.TorchTrainer — Ray 2.35.0," docs.ray.io. https://docs.ray.io/en/latest/train/api/doc/ray.train.torch.TorchTrainer.html (accessed Aug. 20, 2024).