Python Scaling Frameworks for Big Data Analysis and Machine Learning: A Comparison of Ray and PyTorch

Olga Giannouchou
ECE
NTUA
Athens, Greece
el19613@mail.ntua.gr

Aikaterini Stavrou
ECE
NTUA
Athens, Greece
el19932@mail.ntua.gr

Abstract—The rapid expansion of data volume and complexity has amplified the need for scalable and efficient frameworks for distributed computing in machine learning and big data applications. This paper presents a comprehensive comparative analysis of two widely adopted Python-based frameworks, Ray and PyTorch, focusing on their efficiency in parallelizing computations and scaling across distributed clusters. We evaluate the performance characteristics of both frameworks through benchmarking experiments, examining factors such as training speed, scalability, resource management, and memory efficiency in real-world machine learning tasks. Additionally, we address key challenges such as load balancing, fault tolerance, network latency, and overall system responsiveness. By providing an in-depth comparison of the trade-offs between Ray and PyTorch in various distributed environments, our research aims to offer practical insights for researchers and practitioners. This will assist in selecting the most appropriate framework based on specific computational and operational requirements, thereby enhancing decision-making processes in the development of large-scale machine learning models and data analysis workflows.

Index Terms—python, scaling frameworks, ray, pytorch, big data analysis, machine learning, system responsiveness, parallel computing, resource management

I. System and Software Overview

A. Virtual Machines [1], [2], [3], [4]

In this project, we will utilize Virtual Machines provided by Okeanos-Knossos to carry out our experiments. Virtual Machines offer flexibility, efficiency and scalability.

- **Isolation:** VMs provide a high level of isolation between different applications and services running on the same physical hardware. This isolation enhances security by minimizing the risk of one application affecting or compromising another.
- **Consistent Environments:** VMs allow for the creation of consistent and reproducible environments throughout a project's lifecycle, helping to avoid compatibility issues between development, testing, and production stages, and ensuring smoother deployments.
- **Resource Efficiency:** VMs enable efficient resource utilization by allowing multiple virtual instances to run on a single physical server. This leads to better performance and faster response times for applications and services.
- **Scalability:** VMs facilitate scalability by allowing dynamic adjustment of resources allocated to each VM based on changing project requirements. This adaptability ensures that a project can scale up or down efficiently to handle varying workloads.
- **Cost Savings:** Using virtual machines can lead to cost savings by reducing the need for physical hardware and associated maintenance costs. It allows for better utilization of existing hardware and provides a more cost-effective solution.

B. Python [5], [6]

Python has firmly established itself as a leading language in the realms of big data analysis and machine learning, thanks to its flexibility, comprehensive libraries, and an active developer community. Here are some key aspects that highlight Python's strengths:

- **Machine Learning Frameworks:** Python is home to some of the most widely used machine learning frameworks, including TensorFlow and PyTorch. These frameworks facilitate the development, training, and deployment of complex machine learning models, covering a broad spectrum of applications from computer vision to natural language processing.
- **Extensive Libraries:** Python boasts a rich ecosystem of libraries such as NumPy, pandas, and scikit-learn, which are indispensable for data manipulation, analysis, and machine learning model development. These libraries provide robust tools that simplify complex data tasks, making Python a powerful tool for data scientists.
- **Community Support:** Python's large and active community plays a vital role in the continuous development and adoption of tools for big data analysis and machine learning. This community-driven approach results in frequent updates, extensive documentation, and a wealth of tutorials and resources.
- **Data Visualization:** Python excels in data visualization, with libraries like Matplotlib, Seaborn, and Plotly offering powerful tools for creating insightful visualizations. Effective data visualization is crucial for understanding complex patterns and trends in big datasets.
- **Scalability with PyTorch and Ray:** Frameworks like PyTorch and Ray enable Python developers to scale their applications efficiently. These tools support parallel and distributed computing, allowing users to process large datasets and execute machine learning tasks across clusters of machines. This scalability is crucial for handling big data projects.
- **Interoperability**: Python integrates seamlessly with other technologies and languages, making it an ideal choice for building end-to-end data pipelines. It can connect with databases, cloud services, and other programming languages, allowing for smooth transitions between different stages of data processing and analysis.
- **Education and Accessibility:** Python's clear and concise syntax makes it accessible for beginners, contributing to its popularity in educational settings. This has led to a growing pool of data scientists and engineers proficient in Python.

In summary, Python's versatility, extensive library support, scalability through frameworks like PyTorch and Ray, and a thriving community make it a preferred language for big data analysis and machine learning. Its ecosystem empowers developers and data scientists to tackle complex problems, analyze vast datasets, and build sophisticated machine learning models with ease.

C. Ray [7], [8]

Ray is an open-source, unified framework designed to scale Python and AI applications, including machine learning tasks. By providing an abstraction layer for parallel computing, Ray simplifies the execution of distributed machine learning workflows without requiring deep knowledge of distributed systems. Key features of Ray include:

- **Task Parallelism:** Ray allows Python functions to be parallelized across a cluster of machines, turning them into distributed tasks that can run in parallel.
- **Distributed Data Processing:** Ray simplifies the scaling of data processing tasks, offering tools for efficient handling of large datasets.
- **Distributed Memory:** Ray enables data sharing between tasks running on different machines through its distributed memory model.
- **Actor Model:** Ray extends the actor model, making it possible to create stateful

distributed objects that combine both computation and data handling.

- **Ray Libraries:** It includes domain-specific libraries like RLlib for reinforcement learning, Tune for hyperparameter tuning, and Train for distributed training, which leverage Ray's scalability and parallelism.
- **Dynamic Task Graphs:** Ray dynamically generates task dependency graphs, enabling efficient resource management and task scheduling.
- **Scalability:** Ray can be deployed on clusters in various environments, such as Kubernetes, AWS, GCP, and Azure, with seamless scaling and fault tolerance mechanisms in place.

These features make Ray an excellent tool for machine learning practitioners, ML engineers, and distributed systems engineers, as it scales workloads across clusters, facilitates model training, and handles complex distributed computing tasks.

D. PyTorch [9]

PyTorch is a highly flexible deep learning framework known for its dynamic task scheduling, parallel data structures, and seamless integration with existing Python libraries. Key features of PyTorch include:

- **Dynamic Task Scheduling:** PyTorch can break down complex computations into smaller tasks and dynamically schedule them for parallel execution. This flexibility is crucial for handling large datasets and optimizing performance in memory-constrained environments.
- **Parallel Data Structures:** PyTorch's parallel data structures, such as Tensors, are built for scalability, supporting efficient computation across CPUs, GPUs, and even distributed clusters, allowing operations to scale beyond the limitations of a single machine.
- **Integration with Existing Libraries:** Integration is a core strength of PyTorch, as it works seamlessly with popular Python libraries like NumPy, pandas, and scikit-learn, enabling users to parallelize and distribute their existing codebases with minimal modifications. This ease of integration makes PyTorch a popular choice among data scientists for machine learning workflows.
- **Support for Out-of-Core Computing:** PyTorch supports out-of-core computing, managing datasets larger than memory by dynamically loading and unloading data, which is essential for big data processing.
- **Scalability:** PyTorch's scalability makes it highly versatile, capable of running on anything from a single laptop to large-scale clusters, making it suitable for both research and production environments. The framework's adaptability and performance enhancements in recent versions (e.g., torch.compile in PyTorch 2.0) continue to strengthen its position in AI development.

PyTorch excels in dynamic task scheduling, scalability, and deep integration with Python libraries, offering robust support for large datasets and distributed computing. Its continuous innovations ensure that it remains a top choice for AI research and real-world applications.

II. Installation and Setup

A. Prerequisites

For this project, we set up three virtual machines to thoroughly explore parallel data processing. The first machine functioned as the master, while the other two acted as workers. We selected Ubuntu 16.04.3 LTS, allocating 4 CPU cores, 8 GB of RAM, and 30 GB of storage to each machine. Additionally, we configured the development environment, including Python and the Ray and PyTorch frameworks.

B. Ubuntu

To begin, we need to update the Linux environment. This is achieved by running the following commands in the terminal, which will ensure that the system's packages are up-to-date:

```
sudo apt update
sudo apt upgrade
```

C. Python

To set up Python, we run the following commands in the terminal:

```
sudo apt install python3
python3 --version
```

To ensure the Python environment is properly configured, we proceed with installing pip, which is necessary for installing additional frameworks:

```
sudo apt install python3-pip
```

Since Ray and PyTorch rely on numpy, we install it with the following command:

```
python3 -m pip install -U numpy
```

Additionally, numpy requires sciPy, so we install it as well:

```
python3 -m pip install -U sciPy
```

For machine learning applications, we also install scikit-learn, lightgbm, xgboost and pandas:

```
python3 -m pip install -U scikit-learn
python3 -m pip install -U lightgbm
python3 -m pip install -U xgboost
python3 -m pip install -U pandas
```

D. Ray

After successfully installing Python in our system we install the Ray framework with the following command:

```
python3 -m pip install -U ray
```

E. PyTorch

Finally, we install the PyTorch framework:

```
pip3 install torch torchvision torchaudio
```

F. Networking

An internet connection is required between the machines for the work distribution to occur. In real-world data center scenarios, the devices are usually connected via Ethernet through hubs and network switches that divide the center and give it structure and hierarchy. Since the computers in this instance are virtual, they are assigned public IPv6 addresses and put in the same NAT network so they can still have access to the internet and communicate with one another.

Extra security precautions are not necessary because the data generated and used by this project is neither sensitive or high risk, but ideally the worker computers should not be able to be accessed from public addresses to reduce the number of potential attack points.

III. Subjects of Analysis

A. Classification [10] [11]

Classification is a supervised machine learning method where the model tries to predict the correct label of a given input data. In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on new unseen data. Machine learning classification can be used in various industries to teach machines where and how to organize massive amounts of data. It is simpler to comprehend and identify data trends once the data has been grouped according to the criteria. Making more precise, data-driven decisions becomes easier after patterns are established. Classification divides vast volumes of data into individually smaller values to facilitate prediction, such as true and false or predefined output labels. This is because different machine learning algorithms cluster information together into specific categories.

IV. Data generation and Data loading

For the data loading part, we decided to generate our own data through a Python script. Because of the flexibility in creating datasets of any size using our custom made script, we were able to generate data and select the most appropriate data size for our experiments. Despite being produced by random number selection, this data has been parameterized to

match actual datasets. With this method, the data can be utilized as naturally occurring data sets, giving a genuine depiction of the settings in which the project will be carried out. This method of creating data aims to improve both the experiment's flexibility and control and the outcomes' usefulness and accuracy.

A. Dataset for Classification

We created a script to generate synthetic classification data and save it in CSV format. The script accepts three parameters: the number of samples (data points), the number of features per data point, and the number of classes or labels. Due to the potential size of the datasets exceeding available system RAM, the data can be generated in smaller chunks, which are then concatenated into a single CSV file. Additionally, a metadata file is produced to provide an overview of the dataset. The script allows for the configuration of chunk size to handle larger datasets efficiently and generates both the dataset and metadata based on the specified size requirements:

	1GB	2GB
Samples	5 M	10 M
Features	10	10

A. Ray

1) Master: After having successfully set up the Ray framework in the master VM, we enter the following command in the master's terminal in order to start the Ray head node and get the URL which will be used in connecting the rest of the worker nodes:

```
ray start --head                                    1
```

2) Workers: After having successfully set up the Ray framework on each worker machine and the Ray head node is already running, through the terminal of each worker we execute this command:

```
ray start --address <IP>:6379                       2
```

By doing this, a Ray worker process that connects to the IP provided will start (in our case the master's IP in our private network).

B. PyTorch

1) Master: After having successfully set up the PyTorch framework, in order to run our scripts in parallel using the both the master and worker nodes, we enter the following command to the master's terminal:

```
torchrun --nproc_per_node=1 --nnodes=3 --node_rank=0 --
master_addr="<IP>" --master_port=<port> <script_name>      1
```

This command sets up a distributed training session using PyTorch across three nodes, with the current node acting as the master (node rank 0). It specifies that the script will run with one process per node.

2) Workers: After having successfully set up the PyTorch framework on each worker machine and ensuring the master node is running, we execute the following commands on each worker node's terminal:

For the first worker:

```
torchrun --nproc_per_node=1 --nnodes=3 --node_rank=1 --
master_addr="<IP>" --master_port=<port> <script_name>      2
```

For the second worker:

```
torchrun --nproc_per_node=1 --nnodes=3 --node_rank=2 --
master_addr="<IP>" --master_port=<port> <script_name>      3
```

These commands initiate distributed training on all three nodes in the cluster, with each node running the same Python script. Node ranks are set as 0 for the master, 1 for the first worker, and 2 for the second worker. The commands also specify that each node will run one process and connect via the specified port.

VI. CODE FOR MEASURING PERFORMANCE

Since Ray and PyTorch are scaling frameworks, they do not affect the majority of the python code we want to execute. Apart from the different methods to connect to the clusters, there are some notable differences in some minor parts of the code.

A. Notable code differences

1) Data handling and loading:

- *Ray:* Data is loaded from a CSV file into a Pandas DataFrame, which is then stored in Ray's object store using *ray.put()* to facilitate data sharing across distributed processes.
- *PyTorch:* Data is also loaded from a CSV file into a Pandas DataFrame, but it is directly split and processed across distributed nodes. Each node receives its own data partition for distributed training.

2) Distributed Training and Parallelism:

- *Ray:* Ray handles distributed training using the *tune.run()* function, which manages trial runs and resource allocation. Data is fetched from Ray's object store, and the training function runs trials in parallel for model training and evaluation.
- *PyTorch:* PyTorch uses libraries *torch.multiprocessing*, *torch.distributed* for parallelism. It initializes distributed training via the *init_process_group* and assigns each node a rank, allowing algorithms to run separately on each node with partitioned datasets.

3) Model Training and Evaluation:

- *Ray:* The model is trained using one of our five machine learning algorithms, with the configuration passed through Ray Tune. The training is done across trials, and results like accuracy are collected after each trial, with the best configuration and accuracy being reported.
- *PyTorch:* The model is trained using one of our five machine learning algorithms as well, but the data is partitioned across distributed nodes. Each node runs its portion of the training independently, and the results (accuracy) are reported by each process. No explicit hyperparameter tuning is performed within the PyTorch setup.

4) Parallel Execution:

- *Ray:* Parallelism is achieved using Ray's distributed framework with trial-based execution, where each trial can be tuned and evaluated separately.
- *PyTorch:* Parallelism is handled through node-level distributed training, where each node performs its own data partition training, without explicit trial-based execution.

B. Connecting to the clusters

1) Ray: In the beginning of the file that contains the python code we want to execute, we need to add the following:

```
ray.init(address="<IP>:6379")
```

Now, the process that is responsible for running the python script will connect to the Ray cluster to distribute the tasks. Between the brackets is the IP address of the machine where the master node is running. At the end of the file we add:

```
ray.shutdown()
```

To close the connection to the head node.

2) PyTorch: In the beginning of the file that contains the Python code we want to execute, we need to initialize the distributed process group to enable communication between nodes. This is done by adding the following line:

```
dist.init_process_group(backend="gloo", rank=<rank>,
world_size=<world_size>)
```

This command connects the current node to the PyTorch distributed cluster. The rank represents the rank of the current node (e.g., 0 for master, 1 for the first worker), and *world_size* refers to the total number of nodes involved in the training.

At the end of the file we add:

```
dist.destroy_process_group()
```

This closes the connection to the PyTorch distributed process group, allowing the resources to be freed for future processes.

VII. Results

Our project's initial aim was to evaluate the effectiveness of big datasets of 3, 5, 10GB. However, due to hardware limitations, we limited the size of the datasets to 1GB and 2GB. This adjustment allowed us to concentrate on comparing the efficiency and accuracy of the Ray and PyTorch frameworks with manageable dataset sizes.

A. Classification

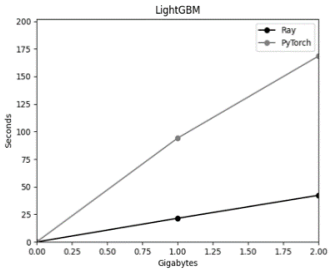

Figure 1: LightGBM - Computational time

Time		
	1GB	2GB
Ray	21.47	42.3
PyTorch	94.1	168.5
Accuracy		
	1GB	2GB
Ray	0.84	0.77
PyTorch	0.84	0.77

Table 1: LightGBM - Time and Accuracy

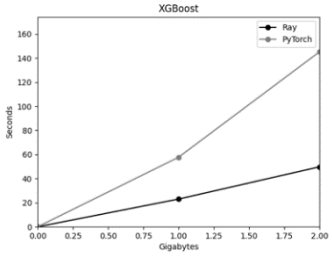

Figure 2: XGBoost - Computational time

Time		
	1GB	2GB
Ray	23.0	49.7
PyTorch	57.7	145.1
Accuracy		
	1GB	2GB
Ray	0.897	0.838
PyTorch	0.897	0.838

Table 2: XGBoost - Time and Accuracy

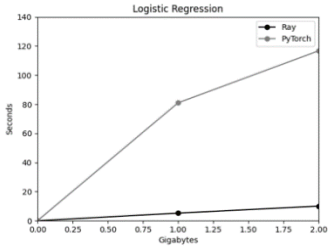

Figure 3: Logistic Regression - Computational time

Time		
	1GB	2GB
Ray	5.22	10.0
PyTorch	80.98	116.7
Accuracy		
	1GB	2GB
Ray	0.58	0.53
PyTorch	0.58	0.53

Table 3:Logistic Regression - Time and Accuracy

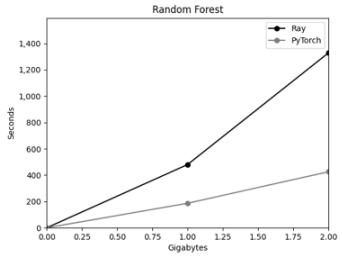

Figure 4: Random Forest - Computational time

Time		
GB/Sec	1GB	2GB
Ray	480.3	1328.2
PyTorch	186.5	425.7
Accuracy		
GB/Ac	1GB	2GB
Ray	0.96	0.95
PyTorch	0.96	0.95

Table 4: Random Forest - Time and Accuracy

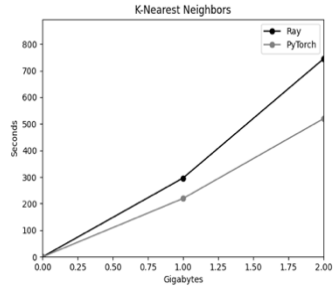

Figure 5: KNN - Computational time

Time		
GB/Sec	1GB	2GB
Ray	296.3	744.3
PyTorch	219.45	519.5
Accuracy		
GB/Ac	1GB	2GB
Ray	0.96	0.96
PyTorch	0.96	0.96

Table 5: KNN - Time and Complexity

B. Accuracy

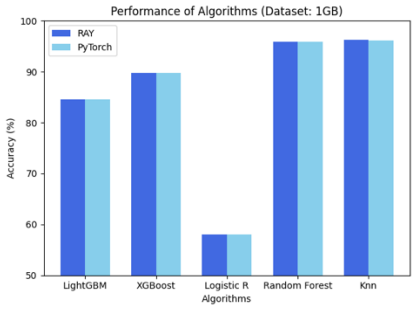

Figure 6: Accuracy with Dataset1

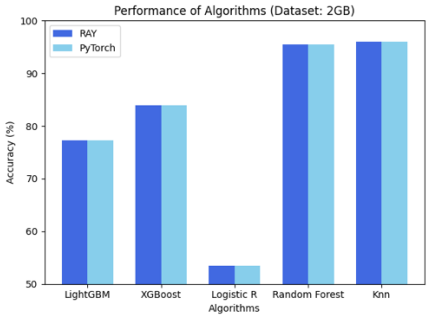

Figure 7: Accuracy with Dataset2

Accuracy		
Algorithm	1GB	2GB
LightGBM	0.84	0.77
XGBoost	0.89	0.84
LR	0.58	0.53
RF	0.96	0.95
Knn	0.96	0.96

Table 6: Accuracy of Algorithms to Data Size

VIII. Observations

A. Classification

1) LightGBM: When it comes to computational efficiency, LightGBM exhibits a notable distinction in execution time when compared to PyTorch, with Ray consistently demonstrating superior processing capabilities. However, it is important to note that despite this discrepancy in performance, both frameworks achieve identical levels of accuracy.

2) XGBoost: Ray is the most efficient choice for XGBoost since it executes substantially faster than PyTorch, especially when working on larger datasets. Both frameworks achieve the same accuracy, so Ray would be the better choice without compromising the model's performance.

3) Logistic Regression: Ray offers a substantial advantage in running time over PyTorch, completing tasks significantly faster. Despite its speed advantage, the accuracy of Logistic Regression is relatively lower compared to other models.

4) Random Forest: In terms of running time, PyTorch performs substantially better than Ray for Random Forest, especially with larger datasets where Ray is notably slower. Regardless of this performance gap, the accuracy remains identical across both frameworks.

5) K-Nearest Neighbors: PyTorch offers superior time efficiency compared to Ray without sacrificing model performance. Both frameworks achieve the same high level of accuracy, making PyTorch the better choice for faster execution while maintaining consistent results.

IX. Conclusion

Ray and PyTorch both offer significant versatility and scalability, making them powerful tools for various machine learning and big data tasks. Our results show that while both frameworks excel in different areas, there are notable differences in efficiency depending on the specific algorithm. For instance, Ray generally outperforms PyTorch in terms of execution speed for algorithms such as LightGBM and XGBoost, whereas PyTorch is faster for Random Forest and K-Nearest Neighbors.

However, it is important to note that PyTorch is more suited for custom machine learning applications, particularly those involving deep learning and neural networks. PyTorch's flexibility and support for dynamic computation graphs make it ideal for tasks that require building and fine-tuning complex neural network architectures, which are less efficiently handled by Ray. Ray, on the other hand, excels in managing parallelism and distributed computing, making it particularly advantageous for scenarios requiring efficient handling of multiple processes, such as large-scale hyperparameter tuning or when working with algorithms like XGBoost.

In conclusion, given the variance in performance across different algorithms and use cases, users and developers should carefully evaluate their specific

requirements. For deep learning tasks, PyTorch is the clear choice, while Ray may be preferable for large-scale distributed computing with non-neural network algorithms. Extensive testing of both frameworks is recommended to determine the most suitable option for specific applications.

X. GITHUB

https://github.com/olgagnn/InformationSystemsProject

REFERENCES

[1] "Okeanos-Knossos Home" https://okeanosknossos.grnet.gr/home

[2] "GRNET Okeanos Services" https://grnet.gr/en/services/cloud-services/okeanos/

[3] "What Are Virtual Machines (VMs)? Use Cases and Benefits" https://konghq.com/blog/learning-center/virtual-machines

[4] "12 Benefits of Virtualization in Cloud Computing" https://www.digitalocean.com/resources/articles/benefits-of-virtualization

[5] "Python's Role in Big Data and Analytics" https://learnpython.com/blog/python-in-big-data/

[6] "How to deal with Big Data in Python for ML Projects (100+ GB)?" https://www.machinelearningplus.com/python/how-to-deal-with-big-data-in-python/

[7] "Ray Overview" https://docs.ray.io/en/latest/ray-overview/index.html

[8] "Distributed Processing using Ray framework in Python" https://www.datacamp.com/tutorial/distributed-processing-using-ray-framework-in-python

[9] "PyTorch Documentation." https://pytorch.org/

[10] "What Is Machine Learning Classification?" https://www.coursera.org/articles/machine-learning-classification

[11] "Classification in Machine Learning: An Introduction" https://www.datacamp.com/blog/classification-machine-learning