Comparing Ray and PyTorch for Scalable Big Data Analytics and Machine Learning

Αντώνιος Αλεξιάδης (03120167)
School of Electrical & Computer Engineering
National Technical University of Athens (NTUA)
Athens, Greece
el20167@mail.ntua.gr

Χαρίδημος Παπαδάκης (03120022)
School of Electrical & Computer Engineering
National Technical University of Athens (NTUA)
Athens, Greece
el20022@mail.ntua.gr

Νικόλαος Μπόθος Βουτεράκος (03120158)
School of Electrical & Computer Engineering
National Technical University of Athens (NTUA)
Athens, Greece
el20158@mail.ntua.gr

Team ID: 21

***Abstract*—Scaling big data analytics and machine learning (ML) workloads efficiently is a fundamental challenge in modern computing. This work compares two prominent Python scaling frameworks, Ray and PyTorch, evaluating their performance across diverse computational tasks. Ray provides a unified framework for distributed computing and data processing [5], [6], while PyTorch is widely used for ML workloads, including distributed training [9], [10].**

Our study examines the scalability and computational efficiency of both frameworks across different types of data-intensive workloads, including graph analytics, clustering, and deep learning pipelines. By systematically varying dataset sizes, computational resources, and workload characteristics, we assess their relative strengths and weaknesses. The results provide insights into their suitability for different big data and ML applications, highlighting trade-offs in performance, ease of use, and scalability.

***Index Terms*—Big Data Analytics, Distributed Computing, Machine Learning, Ray, PyTorch, Parallel Computing, Clustering, Scalability, Performance Evaluation.**

I. Introduction

The rapid growth of big data and machine learning (ML) applications has driven the need for scalable frameworks capable of efficiently processing large datasets and executing computationally intensive tasks [5], [9]. Traditional single-node execution is often insufficient to handle the increasing demands of modern data pipelines, necessitating the adoption of distributed computing frameworks.

Ray and PyTorch are two prominent frameworks that address these challenges. Ray is a general-purpose distributed computing framework designed to scale Python-based workloads, including data analytics, machine learning, and reinforcement learning [5]. It provides a high-level API for task scheduling, parallel execution, and distributed data processing, making it suitable for large-scale applications. On the other hand, PyTorch, primarily known as a deep learning framework, also offers robust distributed training capabilities and is widely used for ML workloads, including clustering, classification, and predictive modeling [9], [10].

This work aims to compare the performance of Ray and PyTorch in handling large-scale computational tasks. Specifically, we evaluate their scalability and computational efficiency across three distinct workloads: graph analytics, clustering, and deep learning pipelines. The goal is to analyze how each framework performs under different data sizes and computational resources, highlighting their strengths and limitations in real-world scenarios.

To conduct this comparison, we implement a series of experiments where datasets of varying scales are processed using both frameworks. The evaluation focuses on execution time, scalability trends, and system responsiveness under different configurations. By systematically analyzing these aspects, we provide insights into the trade-offs between Ray and PyTorch, helping practitioners select the most suitable framework for their specific big data and ML applications.

To facilitate reproducibility and further exploration, we provide our implementation, datasets, and experiment scripts in an open-source repository [13].

II. Infrastructure and Software Overview

This section provides a detailed description of the computational infrastructure and software stack employed in our study, encompassing the virtualized environment, distributed computing frameworks, and storage solutions.

A. Computational Infrastructure

Our experiments were conducted using virtual machines (VMs) provisioned through the Okeanos-Knossos cloud service, an Infrastructure as a Service (IaaS) platform developed by GRNET for the Greek Research and Academic Community

[4]. This platform enables users to create VMs with customizable specifications, facilitating the deployment of tailored virtual environments suited to our computational requirements.

Each VM was configured with multi-core CPUs and adequate memory to support the demands of distributed processing tasks.

B. Software Stack

The software stack utilized in our study comprises the following components:

1) Ray: Ray is an open-source distributed computing framework designed to scale Python applications, particularly in machine learning and data analytics [5], [6]. It offers a high-level API for task scheduling, parallel execution, and distributed data processing, enabling efficient handling of large-scale computational tasks.

2) PyTorch: PyTorch is a widely-used open-source machine learning framework known for its flexibility and ease of use. It supports both single-node and distributed training through its `torch.distributed` package, which provides communication primitives for multiprocess parallelism across multiple computation nodes [9], [10].

3) Hadoop Distributed File System (HDFS): For data storage, we employed the Hadoop Distributed File System (HDFS), a distributed file system that provides high-throughput access to application data and is designed to scale from single servers to thousands of machines [7], [8]. HDFS was utilized to store large datasets, ensuring efficient data access and management across our distributed computing environment.

C. Integration Strategy

In our experimental setup, Ray and PyTorch were employed to perform parallel and distributed processing tasks. This approach allowed us to leverage the strengths of each framework in handling large-scale computational workloads.

III. Environment Setup

This section provides a detailed overview of the environment setup, including the virtual machines (VMs), network configuration, cluster configuration, and software installations.

A. Virtual Machine Characteristics

The computational environment for our experiments was deployed on virtual machines (VMs) provisioned through the Okeanos-Knossos cloud service. Each VM was configured with sufficient resources to support the execution of distributed computing and machine learning workloads. The chosen specifications balance computational power, memory capacity, and storage to ensure efficient performance while maintaining scalability.

All VMs were based on the Ubuntu Jammy Cloud LTS image, specifically Ubuntu 22.04.3 LTS [14], a long-term support release providing stability and security updates.

Hardware Specifications:

- CPU: 4 CPUs
- Memory: 8 GB RAM
- Storage: 30 GB Disk

B. Networking Configuration

To enable efficient and secure communication between virtual machines (VMs), the networking setup was designed according to the constraints of the Okeanos-Knossos cloud infrastructure [3]. The setup ensures VM2, which lacks a public IP, can access the internet via VM1, and VM3, hosted in a separate Okeanos account, can connect securely through a VPN.

The key elements of the setup include:

- A private network shared between VM1 and VM2, facilitating internal communication.
- VM1 acting as a NAT gateway, forwarding VM2's traffic to the internet.
- VM3, which exists in a different account, accessing VM1 via a WireGuard VPN.

1) VM Roles and Network Configuration: Each VM was assigned specific roles to maintain an efficient networking structure:

- VM1 (Gateway and NAT Server): Holds a public IP (`83.212.76.26` on `eth2`) and a private IP (`192.168.0.1` on `eth1`). It performs NAT to enable VM2's internet access and acts as a VPN endpoint for VM3.
- VM2 (Internal Node): Connected to VM1 via the private network (`192.168.0.3` on `eth1`). It routes its internet traffic through VM1.
- VM3 (External Node): Resides in a separate Okeanos account with a public IP (`83.212.76.27`). It connects to VM1 using WireGuard.

2) Private Network Setup for VM1 and VM2: To establish internal communication between VM1 and VM2, both were assigned private IP addresses. Additionally, VM1 was configured as a default gateway for VM2.

First, IP forwarding was enabled on VM1 to allow routing between interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
echo "net.ipv4.ip_forward = 1" >> /etc/sysctl.conf
sysctl -p
```

Then, the private IPs were assigned:

```
# On VM1:
ip addr add 192.168.0.1/24 dev eth1

# On VM2:
ip addr add 192.168.0.3/24 dev eth1
```

VM1 was set as the default gateway for VM2:

```
ip route add default via 192.168.0.1
```

The private network connection was tested by sending ICMP packets:

```
ping 192.168.0.3  # From VM1 to VM2
ping 192.168.0.1  # From VM2 to VM1
```

3) Enabling Internet Access for VM2 via VM1 (NAT Configuration): Since VM2 does not have a public IP, it requires VM1 to perform Network Address Translation (NAT) so it can access the internet. We used `iptables` for NAT masquerading [16], [17].

To set up NAT on VM1, all existing firewall rules were first cleared:

```
iptables -F
iptables -t nat -F
```

Then, NAT masquerading was enabled to translate VM2's internal IP to VM1's public IP when forwarding traffic to the internet:

```
iptables -t nat -A POSTROUTING -o eth2 -j MASQUERADE
iptables -A FORWARD -i eth1 -o eth2 -j ACCEPT
```

To ensure that the changes persist across reboots, the rules were saved:

```
apt install iptables-persistent
netfilter-persistent save
netfilter-persistent reload
```

The NAT configuration was verified by testing internet access from VM2:

```
ping 8.8.8.8
```

4) WireGuard VPN Setup for VM3: Since VM3 is in a separate Okeanos account, it lacks direct access to VM1's private network. To securely connect VM3 to VM1 and VM2, a WireGuard VPN was established [15].

First, WireGuard was installed on VM1:

```
apt install wireguard
```

Then, cryptographic key pairs were generated:

```
wg genkey | tee privatekey | wg pubkey > publickey
```

The WireGuard configuration file for VM1 was created at `/etc/wireguard/wg0.conf`:

```
[Interface]
Address = 10.0.0.1/24
PrivateKey = <VM1_PRIVATE_KEY>
ListenPort = 51820

[Peer]
PublicKey = <VM3_PUBLIC_KEY>
AllowedIPs = 10.0.0.2/32
```

WireGuard was then started on VM1:

```
wg-quick up wg0
```

Similarly, WireGuard was installed and configured on VM3:

```
apt install wireguard
```

The WireGuard configuration for VM3 was saved in `/etc/wireguard/wg0.conf`:

```
[Interface]
Address = 10.0.0.2/24
PrivateKey = <VM3_PRIVATE_KEY>

[Peer]
PublicKey = <VM1_PUBLIC_KEY>
Endpoint = 83.212.76.26:51820
AllowedIPs = 192.168.0.0/24, 10.0.0.0/24
PersistentKeepalive = 25
```

Starting WireGuard on VM3:

```
wg-quick up wg0
```

To allow proper routing of VPN traffic, forwarding rules were added on VM1 (which also relied on proper Linux networking concepts [16]):

```
iptables -A FORWARD -i wg0 -o eth1 -j ACCEPT
iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o
    eth2 -j MASQUERADE
```

Finally, connectivity was tested:

```
ping 10.0.0.1  # From VM3 to VM1
ping 192.168.0.3  # From VM3 to VM2 via VPN
```

5) Final Network Verification: After completing the configuration, several tests were conducted to verify that all components were functioning correctly:

- VM1 and VM2 successfully communicated over the private network.
- VM2 was able to access the internet through VM1.
- VM3 established a secure VPN connection to VM1.
- Traffic from the VPN subnet was correctly routed and allowed internet access.

These steps ensured that VM2 had unrestricted internet access via VM1 while VM3 remained securely connected via VPN.

C. Cluster Setup: Password-less SSH Configuration

To enable seamless communication between cluster nodes, we configured password-less SSH authentication and updated the `/etc/hosts` file on each VM. This setup allows the nodes to communicate using hostnames instead of IP addresses and enables automated SSH connections without requiring passwords.

1) Setting Hostnames on Each Node: Each node in the cluster was assigned a unique hostname for easier identification and communication. The following command was used to set the hostname on each machine:

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

The hostnames assigned were:

- Namenode: `namenode`
- Datanode 1: `datanode1`
- Datanode 2: `datanode2`

2) Updating `/etc/hosts` and Applying Changes: To enable hostname-based SSH access instead of using IP addresses, the `/etc/hosts` file was updated on all nodes with the following mappings:

```
192.168.0.1 namenode
192.168.0.3 datanode1
10.0.0.2   datanode2
```

After updating the file, the network service was restarted to apply the changes:

```
sudo systemctl restart networking
```

This ensures that hostname resolution is applied correctly across the cluster.

3) Generating SSH Keys for Password-less Access: To enable secure password-less SSH between nodes, an SSH key pair was generated on each node:

```
ssh-keygen
```

This command was executed on `namenode`, `datanode1`, and `datanode2`. Each key pair:

- Is stored in `$HOME/.ssh/id_rsa` (private key) and `$HOME/.ssh/id_rsa.pub` (public key).
- Leaves the passphrase empty (when prompted) to enable password-less authentication.

4) Distributing SSH Keys Across the Cluster: The public key from each node was copied to every other node to enable mutual authentication.

On namenode:

```
ssh-copy-id ubuntu@datanode1
ssh-copy-id ubuntu@datanode2
```

On datanode1:

```
ssh-copy-id ubuntu@namenode
ssh-copy-id ubuntu@datanode2
```

On datanode2:

```
ssh-copy-id ubuntu@namenode
ssh-copy-id ubuntu@datanode1
```

5) Verifying Password-less SSH Access: Once the keys were distributed, password-less SSH access was tested from each node:

```
ssh namenode
ssh datanode1
ssh datanode2
```

If the login was successful without a password prompt, the setup was completed correctly.
This password-less SSH setup ensures secure and seamless communication across the cluster, enabling automated distributed processing without requiring manual authentication.

D. Hadoop and HDFS Setup

To enable distributed storage across the cluster, Hadoop and the Hadoop Distributed File System (HDFS) were installed and configured [7], [8]. The setup required one Namenode (master) and two Datanodes (workers). Additionally, Hadoop relies on the Java Virtual Machine (JVM), making it necessary to install and configure Java before deploying Hadoop.

1) Installing Java (JVM) for Hadoop: Hadoop is written in Java and requires the Java Development Kit (JDK) to function properly. The OpenJDK package was installed on all nodes:

```
sudo apt update
sudo apt install openjdk-11-jdk -y
```

The Java version was verified using:

```
java -version
```

To ensure Hadoop recognizes Java, the `JAVA_HOME` environment variable was set by adding the following line to `~/.bashrc` on all nodes:

```
export JAVA_HOME=$(dirname $(dirname $(readlink -f $
    (which java))))
```

The changes were applied using:

```
source ~/.bashrc
```

2) Installing Hadoop on All Nodes: Hadoop was installed on all nodes using the following steps:

- Download and Extract Hadoop: The Hadoop binary package was downloaded and extracted:

```
wget https://downloads.apache.org/hadoop/
common/hadoop-3.3.6/hadoop-3.3.6.tar.gz
tar -xvzf hadoop-3.3.6.tar.gz
mv hadoop-3.3.6 /usr/local/hadoop
```

- Configure Environment Variables: The following lines were added to `~/.bashrc` for all nodes:

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:
$HADOOP_HOME/sbin
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/
hadoop
export JAVA_HOME=$(dirname $(dirname $(
readlink -f $(which
java))))
export CLASSPATH=$(hadoop classpath --glob)
```

  Then, the changes were applied:

```
source ~/.bashrc
```

- Set Up SSH Access: Password-less SSH was required for Hadoop daemons to communicate between nodes. The previously configured SSH setup allowed Namenode to connect to Datanodes without a password.

3) Hadoop Configuration Files: Hadoop requires proper configuration of XML files located in `/usr/local/hadoop/etc/hadoop/`. The following configurations were applied to all nodes.

- Configuring `core-site.xml`: The `core-site.xml` file was updated to specify the Namenode and HDFS default file system.

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://namenode:9000</value>
    </property>
</configuration>
```

- Configuring `hdfs-site.xml`: The `hdfs-site.xml` file was updated to define replication factors and Namenode/Datanode storage locations.

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:///usr/local/hadoop/data/
    namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:///usr/local/hadoop/data/
    datanode</value>
    </property>
</configuration>
```

- Configuring `mapred-site.xml`: The `mapred-site.xml` file was configured to define the execution framework for Hadoop.

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

- Configuring `yarn-site.xml`: The `yarn-site.xml` file was updated to define the YARN resource manager and node manager services.

```
<configuration>
    <property>
        <name>yarn.resourcemanager.hostname</
    name>
        <value>namenode</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services</
    name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

- Configuring the Workers File: Hadoop uses a `workers` file to define which nodes participate as Datanodes in the cluster. This file is located at:

```
cat /usr/local/hadoop/etc/hadoop/workers
```

The file was edited at the Namenode to include the following nodes:

```
namenode
datanode1
datanode2
```

This ensures that Hadoop recognizes all worker nodes when starting the cluster.

4) Formatting the Namenode and Starting Hadoop Services: Once Hadoop was configured, the Namenode was formatted to initialize HDFS:

```
hdfs namenode -format
```

After formatting, the Hadoop cluster services were started:

```
start-dfs.sh
```

5) Verifying the Hadoop Cluster Setup: To ensure that the cluster was running correctly, the following checks were performed:

- Check active nodes in the cluster:

```
    hdfs dfsadmin -report
```

- List files in HDFS (should be empty initially):

```
    hdfs dfs -ls /
```

With these steps completed, Java (JVM) was successfully configured, and Hadoop and HDFS were deployed, allowing the cluster to perform distributed storage and computation.

E. Ray and PyTorch Setup

1) Installing Python and Pip: Python and Pip were installed on all nodes:

```
sudo apt update
sudo apt install python3 python3-pip -y
```

2) Installing Ray: Ray was installed using Pip:

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

3) Starting Ray: Ray was started on the Namenode as the head node:

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

Worker nodes were connected with:

```
ray start --address='83.212.76.26:6379'
```

Cluster status was verified using:

```
ray status
```

4) Ray Dashboard and Remote Access: The Ray Dashboard provides a web-based interface for monitoring the cluster, including node status, resource utilization, and task execution details. It runs by default on port 8265 of the head node.

Since the dashboard is inside the private network, SSH tunneling was used to access it remotely. The following command was executed from the local machine via VM1 (public IP 83.212.76.26):

```
ssh -L 8265:localhost:8265 ubuntu@83.212.76.26
```

Once the tunnel was established, the dashboard was accessible in a browser at:

```
http://localhost:8265
```

5) Installing PyTorch: Since the virtual machines are CPU-only, PyTorch was installed without GPU support:

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

IV. Distributed Execution Methodology

This section describes how distributed processing was organized and executed using Ray and PyTorch. Both frameworks enable parallel execution across multiple worker nodes, but they differ in their execution models, scalability, and task handling.

A. Ray-Based Execution

Ray follows a task-based distributed execution model, where computations are broken down into independent remote tasks that are executed asynchronously across worker nodes [5], [6]. Ray was initialized following the setup described in the previous section, where the Namenode was configured as the head node, and worker nodes were dynamically added to the cluster. The key components of Ray's execution include:

- Ray is only executed on the Namenode. Unlike PyTorch, where each node runs the same script, in Ray, the entire program is executed only on the Namenode:

```
python3 script.py
```

  This command is not run on worker nodes, as Ray automatically schedules tasks across them.
- Scalability: New worker nodes can be added or removed dynamically using:

```
ray start --address='83.212.76.26:6379'
ray stop
```

- Remote Task Execution: Functions intended for distributed execution are marked with `@ray.remote`, allowing them to run on worker nodes.
- Parallel Task Invocation: Remote functions are executed in parallel using `@ray.get`. For example:

```
results = ray.get([preprocess_data.remote(
df_chunk) for df_chunk in data_chunks])
```

- Cluster Termination: Once execution is complete, the Ray session is terminated and the cluster resources are released using:

```
ray.shutdown()
```

Ray's architecture enables efficient workload distribution and fault tolerance, as tasks are scheduled dynamically based on resource availability.

B. PyTorch-Based Execution

PyTorch follows a data-parallel execution model, where each worker node runs an identical copy of the script but processes a distinct subset of data. Unlike Ray, PyTorch requires explicit process coordination across all nodes. PyTorch's distributed execution is structured as follows [9], [10]:

- Replicated Code Execution: The same Python script must be present and executed on every worker node.
- Process Group Initialization: Each node establishes a communication channel with all other participating nodes by executing:

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

  This function initializes the distributed environment, ensuring that:
  - Each process is registered in the system and knows its rank (ID).
  - The master node (`rank = 0`) coordinates execution and typically aggregates results.
  - Worker nodes (`rank > 0`) perform distributed computations.
  - The total number of participating processes is defined by `world_size`.
  - Nodes can communicate efficiently and synchronize computations.

  The function must be executed on every participating node before any distributed operations such as `dist.gather()` are performed. Once execution is complete, the process group should be terminated using:

```
dist.destroy_process_group()
```

- Dataset Partitioning: Data is divided among nodes using `Dataset` and `DataLoader`, ensuring parallel processing:

```
sampler = DistributedSampler(dataset,
num_replicas=world_size, rank=rank)
dataloader = DataLoader(dataset, batch_size=
batch_size, sampler=sampler)
```

- Executing Processes on Each Node: The script is launched with the following command:

```
torchrun \
    --nnodes=3 \
    --nproc_per_node=4 \
    --node_rank=0 \
    --master_addr="namenode" \
    --master_port=29500 \
    script.py
```

  The `nnodes` parameter controls the number of participating nodes, and each node is assigned a unique `node_rank`.
- Result Synchronization: The results of computations are aggregated across nodes using:

```
dist.gather_object(local_results,
global_results)
```

C. Scalability and Node Management

Both Ray and PyTorch allow flexible scalability by modifying the number of participating nodes:

- In Ray, worker nodes can be dynamically added or removed during execution without restarting the cluster.
- In PyTorch, the number of nodes is predefined at launch, requiring explicit rank assignments for each node.

D. HDFS Integration for Data Storage

To handle large-scale datasets, both frameworks were integrated with the Hadoop Distributed File System (HDFS). This allowed worker nodes to process large datasets without loading everything into memory.

- Accessing HDFS: Both Ray and PyTorch accessed data directly from HDFS using `pyarrow`:

```
from pyarrow.fs import HadoopFileSystem
hdfs = HadoopFileSystem(host="namenode",
port=9000)
```

- Reading Data in Batches: Data was read efficiently in chunks:

```
with hdfs.open_input_file("hdfs:///data/
large_dataset.csv") as file:
    csv_reader = pv.open_csv(file)
```

E. Automation with Shell Scripts

To simplify execution, shell scripts were created for PyTorch distributed execution. These scripts handle:

- Environment variable setup for distributed processing.
- Launching `torchrun` across multiple nodes.
- Logging execution details for debugging and monitoring.
- Automating cluster management, reducing manual configuration overhead.

The use of automation scripts streamlined execution across multiple machines, ensuring consistency and reducing setup complexity. The scripts used in this project are available in our GitHub repository [13].

V. Experiments

A. Clustering NYC Yellow Taxi Trip Data

1) Overview: Clustering is a fundamental unsupervised learning technique used to identify patterns and group similar data points. This experiment applies clustering to geospatial data to identify common pickup locations from NYC yellow taxi trip records [1], [2]. Given the large dataset size, distributed processing was required to handle data efficiently across multiple nodes.

A major challenge in distributed clustering was aggregating partial results computed on different nodes. Initially, clustering was performed independently on data chunks distributed across the cluster. The partial results (i.e., cluster centroids and sizes) were aggregated into a final set of clusters to ensure meaningful and coherent groupings.

2) Dataset and Preprocessing: The dataset consists of publicly available NYC taxi trip records capturing millions of rides across the city. Three datasets covering different time periods were used:

- yellow_tripdata_2015-01.csv
- yellow_tripdata_2016-01.csv
- yellow_tripdata_2016-02.csv

Each dataset contains ride details such as pickup/dropoff coordinates, timestamps, trip distances, and fares. Since a major aspect of the experiment was evaluating the capabilities of Ray and PyTorch in executing complex ETL workloads, meaningful data preprocessing was integrated before clustering. The ETL process consisted of:

- Outlier Removal: Unrealistic trips (e.g., unrealistic durations, extreme distances, erroneous coordinates) were filtered based on predefined thresholds derived from prior analysis.
- Feature Engineering: Additional attributes such as trip duration and speed were computed to further contribute to outlier removal and support meaningful clustering.

3) Distributed Clustering Approach: Clustering was performed in two stages:

- Local Clustering: Each node processed a subset of the dataset, performed feature extraction, and applied K-Means clustering.
- Global Aggregation: The computed cluster centers were aggregated across nodes to obtain the final set of clusters.

This approach ensured scalability and avoided excessive communication overhead. The final aggregation step refined cluster centers using a secondary clustering pass, ensuring consistency across nodes.

4) Ray-Based Distributed Execution: Ray was used to parallelize data preprocessing and clustering, leveraging its task-based distributed execution model.

- Cluster Initialization: Ray was initialized on the namenode, with workers dynamically assigned tasks.
- Task Scheduling: Ray tasks were defined using the `@ray.remote` decorator, allowing parallel execution across multiple workers.
- Data Loading: Data was read from HDFS in batches to balance memory usage and processing speed. Each batch was loaded into a Pandas DataFrame and distributed across Ray tasks. The `ray.get()` function was used to retrieve processed batches:

```
preprocessed_chunks = ray.get(
preprocess_tasks)
```

- Feature Extraction: Preprocessing operations (timestamp conversion, trip duration calculation, speed estimation) were executed in parallel across nodes.
- Clustering: Each Ray worker applied K-Means clustering independently on each assigned data chunk using [11]:

```
@ray.remote
def kmeans_cluster(data_chunk, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters,
random_state=42)
    labels = kmeans.fit_predict(data_chunk)
    return labels, kmeans.cluster_centers_
```

- Silhouette Score Calculation: Clustering quality was evaluated using [12]:

```
silhouette = silhouette_score(data_points,
labels) if len(set(labels)) > 1 else -1
```

- Result Aggregation: The computed cluster centroids were gathered from all workers and re-clustered to form the final cluster set.

- Execution Command: The Ray execution was initiated from the namenode using:

```
python3 nyc_taxi_ray_cluster.py --files hdfs
:///data/nyc_taxi/yellow_tripdata_2015-01.
csv
```

5) PyTorch-Based Distributed Execution: PyTorch was used to implement a distributed clustering pipeline using its `torch.distributed` framework.

- Process Group Initialization: Each node joined the distributed execution group using:

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

- Batch Processing: Data was loaded in chunks from HDFS, ensuring efficient memory usage. Each node processed only data batches assigned to its rank using:

```
if idx % world_size == rank:
    df_batch = preprocess_data(batch.
to_pandas())
```

- Dataloader Utilization: PyTorch's DataLoader framework was used to efficiently iterate through the dataset:

```
dataloader = DataLoader(dataset, batch_size=
batch_size, shuffle=False)
```

- Clustering: Each node applied K-Means clustering to its local data.
- Silhouette Score Calculation: Similar to Ray, PyTorch also computed silhouette scores for cluster evaluation.
- Result Aggregation: The node with rank 0 collected and aggregated clustering results using:

```
dist.gather_object(cluster_results,
gathered_cluster_data)
```

- Final Processing: The namenode (rank 0) performed final post-processing, such as writing the results to output files and generating cluster visualization plots.
- Execution Command: The PyTorch execution was automated using shell scripts and started using:

```
torchrun --nnodes=3 --nproc_per_node=4 --
node_rank=0 \
--master_addr=namenode --master_port=29500
nyc_taxi_pytorch_cluster.py
```

6) Summary: This experiment demonstrated the feasibility of distributed clustering using Ray and PyTorch. Both frameworks efficiently handled large-scale geospatial data, enabling parallel execution across multiple nodes.

The next sections will present detailed results, performance comparisons and observations.

B. PageRank on Twitter7 Data

1) Overview: PageRank is a fundamental algorithm used for ranking nodes in a graph based on their relative importance. Originally developed by Google, it assigns a numerical weight to each node, representing its significance within the network. This algorithm is widely applied in search engines, social network analysis, recommendation systems, and citation networks.

A key challenge in distributed PageRank computation is the synchronization and aggregation of rank scores across multiple workers, which must communicate efficiently to ensure the correctness of the final ranking. PyTorch and Ray adopt different approaches to handle this communication, leading to different performance characteristics in execution time, memory consumption, and scalability.

2) Dataset and Preprocessing: The dataset used for this experiment is Twitter7, a large-scale directed graph dataset obtained from the Stanford Network Analysis Project (SNAP) and made available on Kaggle [14]. This dataset represents a Twitter follower network, where nodes correspond to Twitter users and directed edges represent follower relationships. The full Twitter7 graph dataset consists of 17 million users (nodes), 464 million follower relationships (edges) and is 24.66 GB in total size. Due to limited resources the dataset was split in to 3 different subsets:

- 1 GB subset: twitter7_1gb.csv
- 2.5 GB subset: twitter7_2.5gb.csv
- 5 GB subset: twitter7_5gb.csv

Each subset retains the same structural characteristics as the full dataset, but represents a different scenario and varying graph processing requirements while ensuring that PageRank computations across different sizes remain comparable.

During preprocessing, each dataset was divided into manageable chunks. Within each chunk, nodes were remapped to ensure a contiguous range of node identifiers. This remapping involved creating a mapping of original node IDs to a new set of consecutive integers, which facilitated efficient computation. However, it is crucial to maintain consistency in node identifiers across all chunks to preserve the integrity of the graph structure during distributed processing. Inconsistent node mappings can lead to incorrect PageRank calculations, as the algorithm relies on the accurate representation of the graph's connectivity.

3) Distributed Clustering Approach: Once again clustering was performed in two stages:

- Graph partitioning and chunk-based processing to manage memory constraints and localize computation of PageRank
- Global Rank synchronization and aggregation of results to accumulate scores across all partitions, ensuring correctness.
- In both implementations we used an already implemented version of PageRank available by `torch_ppr`. [15]

4) Ray-Based Distributed Execution: Ray's task-based distribution execution was again leveraged to parallelize data processing and PageRank execution. The cluster initialization, task scheduling and data loading follow the same principles as those explained in Clustering NYC Yellow Taxi Trip Data experiment.

Ray enables automatic parallel execution of `process_chunk_remote()` across multiple worker nodes. The following execution model applies:

- Task Scheduling: This function is called remotely for each chunk using `ray.remote()`, and Ray automatically distributes these tasks across available worker nodes.The Ray task queue ensures that as soon as a worker completes processing one chunk, it moves to the next available chunk, optimizing parallel execution.

```
@ray.remote
def process_chunk_remote(chunk_data):
```

- Execution of Remote tasks: Instead of sequential processing, Ray asynchronously schedules multiple remote tasks

```
process_chunk_remote.remote(chunk)
futures.append(fut)
```

- Efficient parallelism with ray.get(): Once a batch of chunks is processed, the system retrieves the results asynchronously using:

```
partial_dict_list = ray.get(futures)
```

  Allowing Ray to continue processing new chunks while already completed ones are aggregated.
- Intermediate merging: Since PageRank computations generate large amounts of intermediate data to alleviate memory usage, results were merged incrementally to avoid this excessive memory usage.

```
merged_dict = aggregate_dicts(merged_dict,
pd)
merged_dict = top_k(merged_dict, 10000)
save_intermediate_results(merged_dict, f"
chunk_{i+1}
```

- Execution command: Ray execution was once again initiated from the namenode using the following command since the dataset declaration is hardcoded in the code:

```
python3 pagerank-ray.py
```

5) PyTorch-Based Distributed Execution: Using PyTorch's `torch.distributed` framework we create a distributed clustering pipeline.

- Process Group Initialization: We identify the rank and of each worker node joining the execution group and leverage PyTorch's CPU distribution using `gloo`:

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

- Batch Processing: Each worker processes a batch of edges, applying the PageRank computation to its assigned graph partition. We efficiently iterate through the dataset using PyTorch's DataLoader framework.

```
DistributedSampler(dataset, num_replicas=
world_size, rank=rank, shuffle=False)
dataloader = DataLoader(dataset, batch_size
=1024 * 1024, sampler=sampler)
```

- Result Aggregation and Final Processing: After all batches are processed, each worker saves its final results. A global barrier synchronization ensures that all workers finish before proceeding. The ranked 0 worker loads and merges all results.

```
save_intermediate_results(global_results, '
final', rank)
dist.barrier()
if rank == 0: aggregated_results =
aggregate_results() display_results(
start_time, aggregated_results, config)
```

- Execution Command: The PyTorch command used to run this distributed environment was again automated using a shell script and started using:

```
torchrun --nnodes=3 --nnproc_per_node=4 --
node_rank=0 \
--master_addr=namenode --master_port=29500
pagerank.py
```

C. Lesion Classification

1) Overview: Lesion classification is a binary classification task aimed at distinguishing between malignant and benign skin lesions. This experiment utilizes the MRA-MIDAS dataset [18], a multimodal dataset containing both dermoscopic and clinical images along with tabular metadata. Given the large size of the dataset and the computational intensity of feature extraction, distributed processing was required to efficiently handle data across multiple nodes.

The primary computational challenge in this task was extracting feature vectors from images using a pre-trained convolutional neural network (CNN) while maintaining scalability. The dataset was divided into three subsets of increasing size to evaluate the frameworks' ability to scale. In addition, 10-fold cross-validation was implemented to assess model generalization, with fold-wise training distributed across nodes.

2) Dataset and Preprocessing: The dataset consists of systematically paired dermoscopic and clinical images of skin lesions, along with tabular metadata describing patient attributes and lesion characteristics. The dataset was divided into three subsets for experimentation:

- data_1: 1.05GB
- data_2: 2.16GB
- data_3: 3.37GB

Preprocessing involved the following steps:

- Tabular Data Processing: Since the tabular data was relatively small, it was preprocessed on a single node and loaded into memory in its entirety. Minimal transformations were required before integration with image-based features.
- Feature Extraction: Image-based feature vectors were extracted using a ResNet50 model pretrained on ImageNet, with the classification layer removed.
- Data Combination: Extracted feature vectors were combined with preprocessed tabular data to create a unified representation for classification.

3) Distributed Execution Approach: The main computational bottleneck was the extraction of feature vectors from images. Thus, distribution primarily focused on parallelizing this step, while other preprocessing tasks remained centralized. Additionally, 10-fold cross-validation was also distributed across workers to parallelize training.

4) Ray-Based Distributed Execution: Ray was employed to distribute both feature extraction and cross-validation. Given that feature extraction from images using a deep learning model is the most computationally expensive step in the pipeline, it was essential to leverage Ray's parallel task execution. The 10-fold cross-validation process was also distributed to efficiently train and evaluate the classification model.

- Task Scheduling: Ray tasks were defined using the `@ray.remote` decorator, which allowed functions to execute asynchronously on different workers. This enabled parallel execution of both feature extraction and cross-validation folds. The feature extraction process involved loading images, processing them through ResNet50 (without its classification layer), and storing feature vectors. Each fold of the cross-validation was treated as a separate task, distributing the model training process.

```
@ray.remote
def batch_feature_extraction(config, batch,
feature_extractor, hdfs):

@ray.remote
def kfold_cross_validation(config, fold_idx,
 train_data, test_data, cnn_feature_columns)
:
```

- Batch-Based Image Processing: The dataset was divided into batches to ensure efficient memory management and parallelism. Each batch was processed as a separate Ray task, and intermediate results were periodically stored to reduce memory pressure in Ray's object store. The `ray.get()` function was used to retrieve the processed feature vectors from remote workers. To further optimize memory usage, results were stored in a local array at set intervals.

```
batch_futures = batch_feature_extraction.
remote(config, batch, feature_extractor,
hdfs)
futures.append(batch_futures)

if len(futures) >= config["save_interval"]
// config["batch_size"]:
    batch_results = ray.get(futures)
```

- Cross-Validation Distribution: The 10-fold cross-validation process was parallelized by assigning each fold as a separate task using Ray's remote execution. The dataset was split into 10 subsets, where 9 folds were used for training and 1 fold for validation. Each training-validation cycle was run in parallel across available nodes to speed up the evaluation process. The `ray.get()` function was used to retrieve the results from the workers.

```
futures = [
    kfold_cross_validation.remote(config,
fold_idx, final_data.iloc[train_idx],
final_data.iloc[test_idx],
cnn_feature_columns)
    for fold_idx, (train_idx, test_idx) in
enumerate(kf.split(final_data))
]
kfold_results = ray.get(futures)
```

- Execution Command: The Ray-based implementation was executed with a command-line argument specifying the dataset subset to be used. This approach allowed for controlled experimentation with different dataset sizes.

```
python3 lesion_classification_ray.py --
dataset data_{number}
```

5) PyTorch-Based Distributed Execution: PyTorch's `torch.distributed` package was used to implement a distributed pipeline similar to Ray, enabling parallel execution of both feature extraction and cross-validation. Unlike Ray, which follows a task-based parallelization model, PyTorch's approach required explicit rank assignment and synchronization across workers.

- Process Group Initialization: PyTorch initializes a distributed execution environment using `dist.init_process_group`. Each worker is assigned a unique rank, and all nodes participate in a collective training or processing job. This setup allows for coordinated execution of feature extraction and model training across nodes.

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

- Feature Extraction: Similar to Ray, image batches were distributed across available workers based on rank, ensuring an even distribution of workload. Each worker processed a subset of the dataset by extracting feature vectors from images using ResNet50. Unlike Ray, PyTorch required explicit assignment of batches to specific ranks.

```
assigned_batches = [batches[i] for i in
range(rank, len(batches), world_size)]
for batch in assigned_batches:
    batch_results = batch_feature_extraction
(config, batch, feature_extractor, hdfs)
```

- Result Aggregation: Once feature extraction was completed across all ranks, the extracted feature vectors were gathered at rank 0 using `dist.gather_object`. A barrier synchronization was added to ensure all nodes had completed processing before proceeding to the next step.

```
dist.gather_object(feature_vectors,
gathered_fv)
dist.gather_object(image_ids, gathered_ids)
dist.barrier()
```

- Broadcasting Processed Data: After combining feature vectors with preprocessed tabular data, the final dataset was only available at rank 0. To ensure all nodes had access to the updated dataset, PyTorch's `dist.broadcast_object_list` function was used to broadcast the processed data structure to all workers.

```
dist.broadcast_object_list(to_broadcast, src
=0)
```

- Distributed Cross-Validation: Similar to Ray, each fold of the 10-fold cross-validation was assigned to a specific worker. The validation results from all ranks were gathered using `dist.gather_object`, and a final barrier synchronization was enforced to ensure consistency before further processing.

```
dist.gather_object(kfold_results,
gathered_results)
dist.barrier()
```

- Execution Command: The PyTorch implementation was executed using a shell script that automated the process across multiple nodes. Similar to Ray, different dataset subsets and also nodes could be used by modifying the dataset and nodes arguments respectively.

```
torchrun --nnodes=3 --nproc_per_node=4 --
node_rank=0 \
--master_addr=namenode --master_port=29500
lesion_classification_pytorch.py
```

6) Summary: This experiment demonstrated the feasibility of distributed feature extraction and model training for lesion classification. Ray and PyTorch effectively handled large-scale multimodal data processing, enabling parallel execution across multiple nodes. The following section presents performance comparisons and insights gained from these experiments.

VI. Result Analysis and Comparison

A. NYC Yellow Taxi Trip Clustering

1) Overview of Clustering Results: The generated clustering results successfully identified meaningful pickup locations across New York City. The cluster centers aligned well with real-world high-density pickup areas, including major transit hubs such as JFK Airport, LaGuardia Airport, and Midtown Manhattan.

The visualization of cluster centers, produced as HTML maps, confirms the validity of the clustering process.

2) Execution Time and Scalability Comparison: The primary objective of the performance evaluation was to compare how Ray and PyTorch scale with increasing dataset sizes and additional compute nodes. Several factors influenced execution time, including:

- Data partitioning and batch size handling.
- The ability of each framework to efficiently distribute computations.
- Overhead introduced by communication and result aggregation.

Two key comparisons were made:

- Execution time as a function of dataset size.
- Execution time as a function of the number of compute nodes.

Ray exhibited longer execution times for larger datasets, particularly in single-node setups. However, as more nodes were introduced, Ray's execution time improved significantly, demonstrating effective parallelism. PyTorch, on the other hand, generally maintained lower execution times but required smaller batch sizes to avoid Out-Of-Memory (OOM) errors.

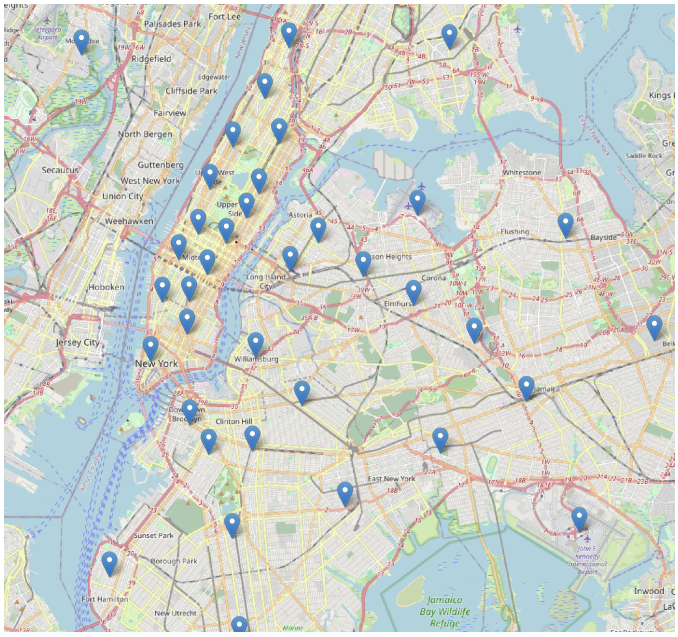

Figure 1. Visualization of Cluster Centers

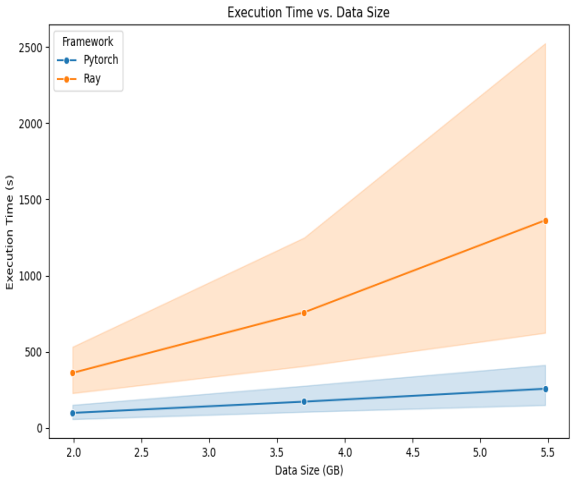

Figure 2. Execution Time vs. Data Size

Batch size was a crucial factor affecting execution time and efficiency. While the same batch size was used for HDFS reading and preprocessing in both frameworks (1MB), a larger number of samples per batch in PyTorch resulted in OOM errors during KMeans clustering. Consequently, the sample size per batch in PyTorch was reduced to ensure stable execution.

This discrepancy impacts execution time comparisons, as PyTorch processed fewer samples per batch than Ray. In theory, Ray's results are superior in terms of clustering accuracy since each task processes a larger amount of data, leading

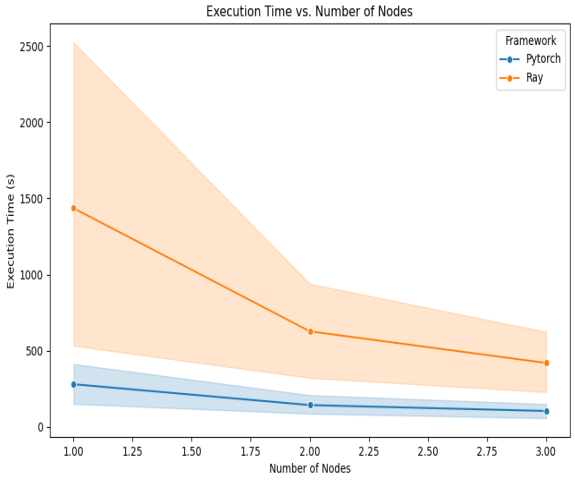

Figure 3. Execution Time vs. Number of Nodes

to better local centroids before global aggregation. However, by reducing the batch size in Ray, we could have potentially achieved better execution times at the cost of clustering quality, as smaller batch sizes would lead to noisier local centroids and less robust final cluster formations.

a) Scaling with Dataset Size: Both frameworks exhibited an increase in execution time as dataset size grew. PyTorch maintained relatively stable execution times due to its structured batch processing, but required smaller sample sizes to avoid Out-Of-Memory (OOM) errors. Ray, in contrast, processed larger data chunks per task, which contributed to its longer execution times, especially for single-node setups. However, as dataset size increased, Ray's task-based execution allowed it to scale efficiently when additional nodes were introduced. While PyTorch consistently executed faster, Ray's ability to process more data per task resulted in higher clustering robustness. Lowering Ray's batch size could have improved execution time but would have led to noisier cluster formations.

b) Scaling with Number of Nodes: Both frameworks demonstrated reduced execution time when more nodes were added. Ray exhibited strong scaling efficiency due to its task-based execution model, significantly reducing execution time as additional nodes were introduced. PyTorch, on the other hand, maintained stable performance across different node configurations, with execution times remaining consistently low. Unlike Ray, PyTorch did not exhibit a major dependency on scaling efficiency, as its execution model inherently avoided excessive overhead per task. However, Ray's ability to distribute tasks dynamically made it particularly well-suited for large-scale workloads, benefiting from parallel execution across multiple nodes.

3) Clustering Quality: Silhouette Score Comparison: To assess the clustering quality, the silhouette score was computed for each experiment. A higher silhouette score indicates well-defined clusters with distinct separation.

Table I
SILHOUETTE SCORE COMPARISON BETWEEN PYTORCH AND RAY

Data Size (GB)	Number of Nodes	PyTorch	Ray
1.99	1	0.3809	0.3793
1.99	2	0.3815	0.3793
1.99	3	0.3811	0.3793
3.7	1	0.3862	0.3860
3.7	2	0.3855	0.3860
3.7	3	0.3839	0.3860
5.48	1	0.3879	0.3885
5.48	2	0.3861	0.3885
5.48	3	0.3847	0.3885

a) Silhouette Score Observations:

- Ray generally achieved slightly higher silhouette scores, particularly for larger datasets.
- PyTorch showed comparable clustering quality but performed slightly worse when the dataset size increased.
- For smaller datasets, both frameworks performed similarly in terms of clustering accuracy.
- The slight improvement in Ray's scores is attributed to its ability to process a larger number of samples per clustering task.

4) Comparative Insights:

a) Advantages of Ray:

- More scalable due to its task-based execution model and much easier to use.
- Able to process larger data chunks per clustering task.
- Generally achieves higher silhouette scores for large datasets.

b) Advantages of PyTorch:

- Faster execution time, especially for smaller datasets.
- Less communication overhead between nodes.
- Works effectively even with fewer compute nodes.

c) Final Observations:

- Both frameworks successfully executed large-scale clustering with meaningful results.
- Ray processed larger sample sizes per task, leading to better clustering quality but longer execution times.
- PyTorch required smaller batch sizes, ensuring stability but increasing the number of iterations.
- Both frameworks demonstrated strong scalability with additional nodes.

B. PageRank

1) Overview of PageRank results: The PageRank execution successfully ranked the most influential nodes within the Twitter7 network, identifying key users with the highest connectivity and influence. The top 10 nodes for each subset consistently exhibited significantly higher PageRank scores compared to the rest of the network, demonstrating their central role in information flow.

These results align with expected network structures, where a small subset of users typically dominates engagement and reach. The high-ranking nodes are likely high-profile accounts

with extensive follower networks, such as public figures, media organizations, or influential entities within the Twitter ecosystem.

2) Execution Time and Scalability Comparison: Below are the results for the execution time as a function of the different number of worker nodes for each distinct subset.

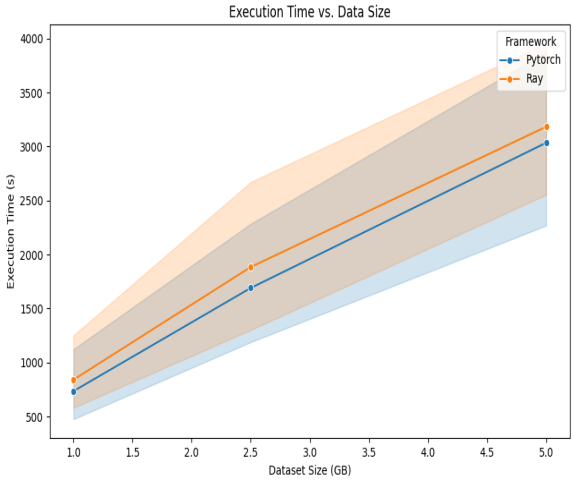

Figure 4. Execution Time vs. Dataset Size

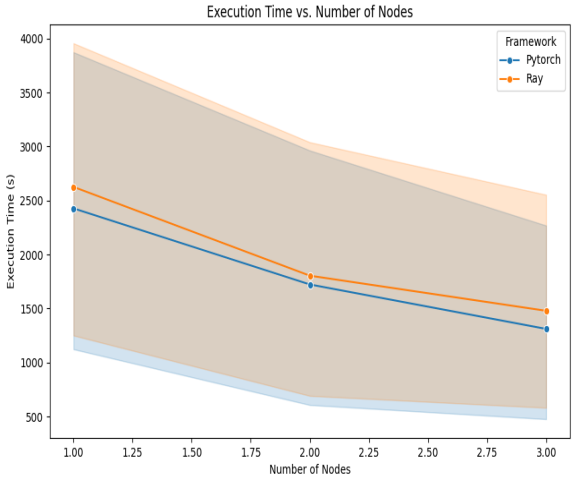

Figure 5. Execution Time vs. Number of Nodes

The execution results reveal a consistent trend: PyTorch outperforms Ray across all dataset sizes and worker configurations. This performance advantage can be attributed to several key factors. PyTorch's torch.distributed framework is optimized for tensor-based computations, allowing it to handle PageRank updates with lower memory overhead compared to Ray's task-based execution model.

The use of `DataLoader` and `DistributedSampler` ensures that only necessary graph partitions are processed by each worker, reducing redundant memory consumption.

PyTorch's Gloo backend enables direct tensor communication between workers, minimizing the overhead introduced by Ray's task scheduling and message-passing model. Ray, while highly flexible, incurs additional communication overhead when workers exchange intermediate results asynchronously.

During the execution of PageRank, both PyTorch and Ray encountered memory constraints when handling larger batch sizes. PyTorch's implementation introduced Out-Of-Memory (OOM) errors with excessively large batch sizes due to the high memory footprint of storing and processing large adjacency matrices in CPU memory. This is because each batch must be converted into a tensor representation, which rapidly increases memory usage as batch size grows.

On the other hand, Ray struggled to handle very large batches, likely due to its task-based execution model, where each worker must independently load, process, and return results. As a result, Ray encountered performance degradation and instability when batch sizes exceeded a certain threshold.

To ensure a fair and meaningful comparison, we standardized the batch size to 30MB for both frameworks, allowing for stable execution while balancing memory efficiency and computational throughput. PyTorch, however, was able to handle a slightly larger batch size of 50MB, which could have further reduced execution time by minimizing the number of iterations and inter-process communication overhead. A larger batch size allows for more edges to be processed per iteration, leading to fewer synchronization points, which is beneficial for PyTorch's synchronous computation model. However, the risk of OOM errors increases as batch size grows, requiring careful tuning based on hardware constraints.

a) Scaling with Dataset Size: When analyzing the scalability of execution across different dataset sizes, we observe that PyTorch's performance advantage is more pronounced for smaller datasets. With 1GB and 2.5GB datasets, PyTorch consistently outperforms Ray due to its lower communication overhead and more efficient batch-based execution. However, as the dataset size increases to 5GB, the execution time difference between the two frameworks narrows. This is because Ray's task-based execution model benefits from better workload distribution, allowing it to overlap computation with data loading and scheduling. While PyTorch remains faster overall, the gap in execution time decreases as Ray's parallelism and dynamic scheduling begin to compensate for its initial inefficiencies.

b) Scaling with Number of Nodes: Both frameworks showed improvements in execution time as the number of nodes increased. Ray demonstrated strong scalability, leveraging its task-based execution model to significantly reduce execution time with additional nodes. PyTorch, in contrast, maintained consistently low execution times across different node configurations, exhibiting less reliance on scaling efficiency. Unlike Ray, PyTorch's execution model inherently minimized per-task overhead, allowing it to remain efficient regardless of node count. However, Ray's dynamic task distribution made

it particularly effective for large-scale workloads, benefiting from parallel execution across multiple nodes.

C. Lesion Classification

1) Overview of Classification Results: The classification results to distinguish between malignant and benign lesions across different dataset sizes and node configurations demonstrate that both frameworks achieved comparable performance in terms of classification accuracy. The primary focus of this comparison lies in evaluating the computational efficiency of Ray and PyTorch when scaling across dataset sizes and distributed nodes.

Key classification metrics, including Accuracy, Precision, Recall, F1 Score, and AUC-ROC, were computed for each experiment. While PyTorch exhibited slightly better classification metrics in most cases, Ray showed competitive performance, particularly in recall, an important factor in ensuring sensitivity to malignant cases.

2) Execution Time and Scalability Comparison: The execution time comparison highlights key differences in how PyTorch and Ray handle distributed deep learning tasks. The following factors influenced performance:

- Efficiency in distributing feature extraction and training workloads.
- Communication overhead and synchronization delays.
- Memory management and batch processing mechanisms.

The execution time was analyzed based on:

- Execution time as a function of dataset size.
- Execution time as a function of the number of compute nodes.

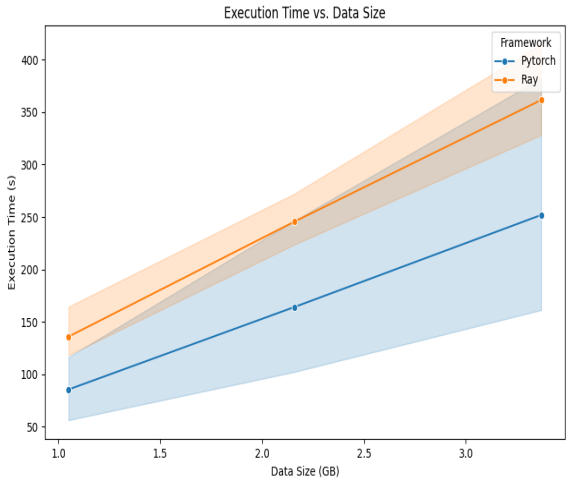

Figure 6. Execution Time vs. Dataset Size

a) Scaling with Dataset Size: As dataset size increased, both frameworks required longer execution times, but PyTorch consistently outperformed Ray in efficiency. Notably:

- For the smallest dataset (1.05GB), PyTorch was significantly faster than Ray in all node configurations.

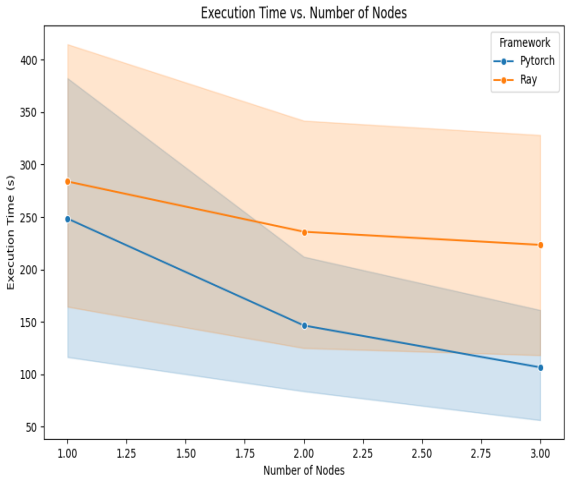

Figure 7. Execution Time vs. Number of Nodes

- For the medium dataset (2.16GB), PyTorch scaled better, maintaining nearly twice the speed of Ray for 2 and 3 nodes.
- For the largest dataset (3.37GB), PyTorch maintained its lead, executing in less than half the time required by Ray when using 3 nodes.

Overall, PyTorch demonstrated superior scalability, efficiently distributing workloads while keeping execution time low.

b) Scaling with Number of Nodes: Both frameworks benefited from additional compute nodes, but PyTorch exhibited a much steeper performance improvement:

- For single-node execution, PyTorch already had an advantage, but Ray's overhead further widened the gap.
- With two nodes, PyTorch saw significant speedup, whereas Ray experienced only modest gains.
- At three nodes, PyTorch completed execution in approximately half the time required by Ray, showcasing its superior parallel efficiency.

Ray's performance degraded as more nodes were added, likely due to higher inter-task communication overhead and inefficient memory handling.

3) Classification Metrics Comparison: To evaluate classification performance, we compared key classification metrics across all dataset sizes and node configurations. The following tables present the mean Accuracy, Precision, Recall, F1 Score, and AUC-ROC for each framework.

a) Observations on Classification Metrics:

- PyTorch achieved higher AUC-ROC in 7 out of 9 cases.
- PyTorch had higher accuracy in 6 out of 9 cases.
- PyTorch had better recall in 7 out of 9 cases.
- PyTorch had better F1 Score in 6 out of 9 cases.
- Precision results were mixed, with Ray leading in 5 cases.

b) Comparison of Overall Classification Performance: Both frameworks delivered comparable classification perfor-

Table II
MEAN CLASSIFICATION METRICS FOR PYTORCH

Data Size (GB)	Nodes	Accuracy	Precision	Recall	AUC-ROC
1.05	1	0.6836	0.7345	0.7033	0.7424
1.05	2	0.6825	0.7343	0.7082	0.7482
1.05	3	0.6781	0.7440	0.6695	0.7458
2.16	1	0.6853	0.6843	0.7574	0.7449
2.16	2	0.6820	0.6993	0.7293	0.7484
2.16	3	0.6880	0.6907	0.7635	0.7520
3.37	1	0.6925	0.7017	0.7038	0.7638
3.37	2	0.6865	0.6932	0.7042	0.7671
3.37	3	0.6925	0.6988	0.7096	0.7688

Table III
MEAN CLASSIFICATION METRICS FOR RAY

Data Size (GB)	Nodes	Accuracy	Precision	Recall	AUC-ROC
1.05	1	0.6891	0.7315	0.7276	0.7502
1.05	2	0.6758	0.7041	0.7578	0.7388
1.05	3	0.6616	0.7007	0.6949	0.7364
2.16	1	0.6815	0.6887	0.7281	0.7410
2.16	2	0.6712	0.6855	0.6986	0.7352
2.16	3	0.6896	0.6963	0.7399	0.7457
3.37	1	0.6981	0.7015	0.7279	0.7697
3.37	2	0.6831	0.7019	0.6886	0.7621
3.37	3	0.6861	0.7107	0.6759	0.7620

mance, with PyTorch exhibiting a slight edge in AUC-ROC, accuracy, and recall. Ray remained competitive, particularly in recall, which is crucial for identifying malignant cases, and showed marginal improvements in precision in certain configurations. However, the differences between the two frameworks were minor, suggesting that execution time is the primary differentiating factor rather than classification quality. Given PyTorch's faster execution while maintaining slightly better performance, it emerges as the more efficient choice for this distributed lesion classification task.

4) Comparative Insights:

a) Advantages of PyTorch:

- Consistently lower execution time across all dataset sizes and node configurations.
- More efficient scaling with additional nodes.
- Slightly better classification performance on key metrics.

b) Advantages of Ray:

- Competitive recall performance, beneficial for detecting malignant cases.
- Flexible task-based execution model.

c) Final Observations:

- Both frameworks achieved similar classification accuracy.
- PyTorch was significantly faster, making it the preferable choice for large-scale deep learning tasks.

VII. CONCLUSION

The comparative analysis of PyTorch and Ray across multiple distributed computing tasks highlights key differences in execution efficiency, scalability, and computational performance. While both frameworks successfully executed the assigned workloads, their performance varied significantly depending on the nature of the task and the underlying distribution strategy.

PyTorch consistently demonstrated superior execution efficiency, achieving significantly lower execution times across all dataset sizes and node configurations. This advantage was particularly evident in large-scale tasks where PyTorch's optimized computation model and memory management allowed for more effective parallelization with minimal overhead. Additionally, PyTorch exhibited slightly better performance in key evaluation metrics, reinforcing its reliability in high-performance deep learning applications.

On the other hand, Ray's task-based execution model proved to be a flexible and scalable alternative, particularly excelling in distributed workload management. However, its execution times were notably higher, often requiring additional nodes to achieve comparable performance to PyTorch. Despite its higher computational overhead, Ray remained competitive in specific metrics, demonstrating robust parallelism and an ability to handle larger data chunks per task. Ultimately, while both frameworks are viable solutions for distributed machine learning, PyTorch emerges as the more efficient choice for tasks requiring both high-performance execution and large-scale data processing.

REFERENCES

[1] NYC YellowTripTaxi (Kaggle Code). [Online]. Available: https://www.kaggle.com/code/elemento/nyc-yellowtriptaxi/notebook [Accessed: Jan. 30, 2025].

[2] NYC Yellow Taxi Trip Data (Kaggle Dataset). [Online]. Available: https://www.kaggle.com/datasets/elemento/nyc-yellow-taxi-trip-data [Accessed: Jan. 30, 2025].

[3] *"How can I access all my VMs using one public IP (NAT)?"*, *Okeanos-Knossos Cyclades User Guide*, GRNET. [Online]. Available: https://okeanos-knossos.grnet.gr/support/user-guide/cyclades-how-can-i-access-all-my-vms-using-one-public-ip-nat/ [Accessed: Jan. 30, 2025].

[4] *Okeanos-Knossos Home*, GRNET. [Online]. Available: https://okeanos-knossos.grnet.gr/home/ [Accessed: Jan. 30, 2025].

[5] *Ray Documentation: A Unified Framework for Scalable and Distributed AI*. [Online]. Available: https://docs.ray.io/en/latest/ray-overview/ [Accessed: Jan. 30, 2025].

[6] *ray-project/ray: Ray is a unified framework for scaling AI and Python applications*, GitHub Repository. [Online]. Available: https://github.com/ray-project/ray [Accessed: Jan. 30, 2025].

[7] *Apache Hadoop*, The Apache Software Foundation. [Online]. Available: https://hadoop.apache.org/ [Accessed: Jan. 30, 2025].

[8] *HDFS Architecture Guide*, The Apache Software Foundation. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html [Accessed: Jan. 30, 2025].

[9] *PyTorch Documentation*. [Online]. Available: https://pytorch.org/ [Accessed: Jan. 30, 2025].

[10] *Distributed Communication Package (torch.distributed)*, PyTorch Tutorials. [Online]. Available: https://pytorch.org/tutorials/intermediate/dist_tuto.html [Accessed: Jan. 30, 2025].

[11] *sklearn.cluster.KMeans Documentation*, scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html [Accessed: Jan. 30, 2025].

[12] *sklearn.metrics.silhouette_score Documentation*, scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html [Accessed: Jan. 30, 2025].

[13] A. Alexiadis, N. Bothos, C. Papadakis, "Comparison between Ray and PyTorch for Distributed Computing," GitHub Repository, 2025. [Online]. Available: https://github.com/harrypapa2002/Comparison-between-Ray-and-Pytorch. [Accessed: Jan. 30, 2025].

[14] Twitter7 Social Network data (Kaggle Dataset). [Online]. Available: https://www.kaggle.com/datasets/wolfram77/graphs-snap-twitter7?select=twitter7.mtx [Accessed: Jan. 30, 2025].

[15] *torch_ppr PageRank documentation*, torch_ppr. [Online]. Available: https://torch-ppr.readthedocs.io/en/latest/usage.html#torch_ppr.api.page_rank [Accessed: Jan. 30, 2025].
[16] Linux Foundation, "Linux Networking Guide," The Linux Documentation Project, 2024. [Online]. Available: https://www.tldp.org/HOWTO/Networking-Overview-HOWTO.html. [Accessed: Jan. 30, 2025].
[17] Iptables Documentation, "Introduction to Linux Firewalling with iptables," Netfilter Project, 2024. [Online]. Available: https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO.html. [Accessed: Jan. 30, 2025].
[18] MRA-MIDAS: Multimodal Image Dataset for AI-based Skin Cancer. [Online]. Available: https://doi.org/10.71718/15nz-jv40 [Accessed: Jan. 30, 2025].