Scalable Distributed Deep Learning with Convolutional Neural Networks using Spark-tensorflow-distributor

Marco Vitucci
Advanced Data Analytics
Big Data Storage and Processing
sba24277

***Abstract*— The rapid growth in data volumes necessitates efficient distributed deep learning methods to achieve practical training times for computationally intensive models such as Convolutional Neural Networks (CNNs). TensorFlow, a widely-used open-source framework, supports distributed training across multiple GPUs, while Apache Spark excels at distributed data processing via parallel computation on commodity hardware. Though Spark lacks native TensorFlow integration, libraries like Spark-tensorflow-distributor enable effective distributed CNN training within Spark clusters. This study implements distributed exploratory data analysis (EDA), preprocessing, and synchronous CNN training using TensorFlow and Spark. Experimental results show a 14% reduction in training time using two executors compared to a single-node setup, with a slight 3.3% accuracy drop due to gradient synchronization overhead. Despite resource limitations, the findings demonstrate the practicality of this approach, offering a scalable and cost-effective solution for distributed CNN training in CPU-based Hadoop/Spark ecosystems with minimal accuracy trade-offs.**

Keywords—Distributed Deep Learning, Tensorflow, Convolutional Neural Networks, Big Data Analytics, Apache Spark, HDFS, CIFAR-10, Spark-tensorflow-distributor, Data Parallelism

I. Introduction

The increase of large datasets across industries has driven the need for scalable machine learning and deep learning techniques (Jordan and Mitchell, 2015). Deep learning, particularly Convolutional Neural Networks (CNNs) for image data, has achieved notable success in tasks like image classification and object recognition (LeCun, Bengio and Hinton, 2015). However, training deep CNN models is computationally intensive and traditionally relies on GPUs due to their capability to efficiently handle extensive matrix operations involved in CNN training (Krizhevsky, Sutskever and Hinton, 2017). Distributed training has emerged as a critical technique to accelerate deep learning by partitioning training tasks across multiple devices or nodes. This approach improves scalability and efficiency by leveraging parallelism, essential for handling large-scale datasets and complex models (Dean *et al.*, 2012). TensorFlow (Abadi *et al.*, 2016), a widely-adopted open-source deep learning framework developed by Google, supports distributed training through various strategies, including MultiWorkerMirroredStrategy (TensorFlow, 2024), which facilitates synchronous training across multiple devices. TensorFlow's distributed capabilities are designed to be flexible, supporting both GPU and CPU-based training across heterogeneous clusters. In contrast, Big Data platforms such as Apache Hadoop and Apache Spark excel at distributed data storage and processing across clusters of commodity hardware (Sewal and Singh, 2021). Merging these two worlds, deep learning and big data frameworks, promises the ability to train models on very large datasets by leveraging cluster resources (Chen and Lin, 2014). Integrating these frameworks enables training directly where data resides, reducing data movement and simplifying infrastructure.

Achieving efficient distributed deep learning on Spark/Hadoop has historically been challenging. Spark (Zaharia *et al.*, 2016) was originally designed for general data-parallel tasks, such as ETL, SQL, and MLlib, and was not initially built with deep learning training in mind. Deep learning frameworks like TensorFlow and PyTorch (Abadi *et al.*, 2016; Paszke *et al.*, 2019) were developed primarily for single-node or small cluster environments, each incorporating their own distributed training mechanisms, complicating their integration with traditional big data stacks. Early efforts to bridge this gap included frameworks such as Yahoo's TensorFlowOnSpark (TFoS) and CaffeOnSpark (Yahoo, 2016, 2017), which enabled distributed deep learning on Hadoop clusters by allowing Spark to handle scheduling of training tasks across nodes. Intel's BigDL (Dai *et al.*, 2019) provided a native Spark library specifically for deep learning tasks, leveraging Intel's Math Kernel Library to optimize CPU cluster performance. More recent solutions include Horovod (Sergeev and Del Balso, 2018), a popular distributed deep learning library utilizing All-Reduce communication, integrated with Spark as Horovod-on-Spark (Horovod, 2019). As part of Project Hydrogen, Apache Spark 2.4 introduced a barrier execution mode (Apache Spark, 2018) designed explicitly to better support the needs of long-running distributed tasks, such as deep learning training. Building upon this, distributed training for TensorFlow was enabled with the Spark-tensorflow-distributor open-source library (TensorFlow, 2021), which leverages Spark's barrier execution mode for launching distributed training tasks. This approach allows users to distribute model training across Spark clusters with minimal modifications to standard single-node TensorFlow code. However, TensorFlow still lacks native support within Spark, relying instead on third-party libraries like Spark-tensorflow-distributor to achieve distributed training functionality. For PyTorch, native support for distributed deep learning was later enhanced with the introduction of TorchDistributor in Spark 3.4 (Apache Spark, 2023). It also utilizes Spark's barrier execution mode to launch training processes on executors, offering a streamlined experience for scaling PyTorch training workloads. Additionally, RayDP (OAP, 2021) provides another approach to distributed TensorFlow training on Spark by integrating Spark with Ray, a framework for distributed computing. RayDP allows Spark and TensorFlow workloads to run within the same Ray cluster, enabling efficient resource sharing and streamlined execution. Despite these advances, challenges

Screencast presentation and datasets in Google Drive:
https://drive.google.com/drive/folders/1wHT8JiTOtUYYTTI291qd-H_EIfXiWyVC?usp=drive_link

remain. Coordination of parallel training workers, efficient sharing of model parameters or gradients, fault tolerance, and seamlessly feeding big data into model training are non-trivial problems (Zaharia *et al.*, 2010; Li *et al.*, 2014).

In this study distributed deep learning training is investigated using TensorFlow integrated with Apache Spark, focusing primarily on enhancing model accuracy and ensuring methodological rigor, rather than optimizing computational throughput. While RayDP offers a hybrid solution by integrating Ray with Spark for distributed training, this study adopts spark-tensorflow-distributor for its simpler integration with existing Spark environments. Unlike RayDP, which requires setting up and managing a separate Ray cluster, spark-tensorflow-distributor allows TensorFlow training jobs to be launched directly on Spark executors, making it convenient for users already leveraging Spark for data processing and distributed computing tasks. Through experimentation is evaluated whether distributed training of CNNs using Apache Spark and TensorFlow can reduce training time without significantly sacrificing accuracy.

The key contributions of this work are summarized as follows:

- Implementing a distributed CNN training pipeline on Spark with data parallelism.
- Performing detailed Exploratory Data Analysis (EDA) and preprocessing on a canonical image dataset (CIFAR-10) in a distributed manner using Spark Core API.
- Evaluating the training outcomes (accuracy, loss, speed) in single-worker vs multi-worker settings (2 executors) using Spark-tensorflow-distributor.
- Analyzing encountered trade-offs, such as communication overhead, gradient synchronization challenges, and fault tolerance.

While CIFAR-10 is a canonical dataset suitable for benchmarking, future work may involve evaluating the proposed approach on larger-scale image datasets (e.g., ImageNet or satellite imagery) to fully stress-test distributed performance under production-scale data loads.

The remainder of this paper is organized as follows. Section 2 reviews related work on distributed deep learning frameworks integrated with Spark, highlighting their architectures, strengths, and limitations. Section 3 outlines the methodology, including system setup, dataset details, and model design. Section 4 presents the results and discussion, analyzing the performance and accuracy trade-offs between single-node and distributed training setups. Finally, Section 5 summarizes the study's findings and proposes potential directions for future research in scaling distributed CNN training on big data platforms.

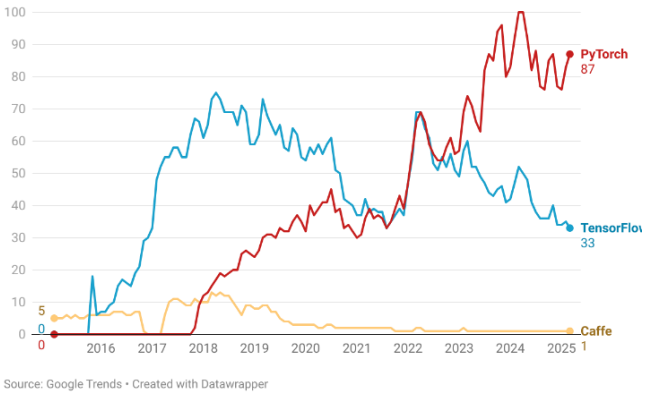

Fig. 1: Google Trends comparison of interest in Caffe, TensorFlow, and PyTorch (2015–2025). Search popularity over time indicates a significant decline in interest for Caffe. In contrast, TensorFlow peaked around 2018–2020 and has gradually declined but still quite used, while PyTorch has seen a steady rise since 2018, surpassing TensorFlow in popularity from 2022 onwards.

II. Related Work

As mentioned in Section 1, distributed deep learning on big data platforms has evolved significantly, with various frameworks offering distinct trade-offs in scalability, ease of use, interoperability, flexibility, robustness, adaptability, and performance. Table I provides a comparative overview of key frameworks, highlighting their core features and how they integrate with Apache Spark.

A. *CaffeOnSpark*

CaffeOnSpark (Yahoo, 2016), developed by Yahoo in 2016, was an early attempt to integrate deep learning with big data platforms. It launched Caffe instances on Spark executors near HDFS data and used AllReduce for gradient synchronization (Feng *et al.*, 2016). However, its complex setup, limited GPU support, and lack of features like dynamic scaling and asynchronous updates, along with its reliance on the now-obsolete Caffe framework, have made it largely irrelevant today. This decline is reflected in public interest trends (see Fig. 1), where Caffe shows minimal activity compared to TensorFlow and PyTorch.

TABLE I. COMPARISON OF DISTRIBUTED DEEP LEARNING FRAMEWORKS INTEGRATED WITH SPARK

Framework	**Features**				
	Communication Method	***GPU Support***	***Native to Spark***	***Data Ingestion Method***	***Training Strategy***
CaffeOnSpark	AllReduce	Limited	Partial	HDFS direct	Synchronous
TensorFlowOnSpark	Parameter Server	Yes	Partial	TFQueue / Spark Feed	Synchronous / Async
BigDL	RDD-based AllReduce	No	Fully native	Spark RDDs	Synchronous
DL4J	Parameter Server	Yes	Java-based	HDFS / Spark	Async / Sync
Horovod-on-Spark	Ring AllReduce (MPI/NCCL)	Yes	Integrated	External ingestion	Synchronous
Spark-tensorflow-distributor	Native AllReduce (Barrier)	Yes	Partial	Spark DataFrames / HDFS	Synchronous
RayDP	Ray actor model	Yes	Hybrid	Spark + Ray Datasets	Synchronous / Async

B. TensorFlowOnSpark (TFoS)

Building on this idea, Yahoo later introduced TensorFlowOnSpark (TFoS) (Yahoo, 2017), adapting TensorFlow's parameter server model for Spark. It supported two data ingestion modes: TFQueue, where TensorFlow natively read from HDFS, and Spark Feeding, where Spark DataFrames/RDDs streamed data into the TensorFlow graph (Yang *et al.*, 2017). Despite its flexibility, TFoS required complex deployment, manual synchronization, and used out-of-band communication, limiting fault tolerance and Spark integration. Additionally, the parameter server model often led to bottlenecks when scaling to large clusters, especially with synchronous training (Sergeev and Del Balso, 2018). As a result, TFoS is now largely obsolete, lacking support for modern TensorFlow workflows.

C. BigDL

BigDL (Dai *et al.*, 2019), developed by Intel, was designed as a Spark-native deep learning framework optimized for CPU clusters. By leveraging Spark's RDD model and Intel's Math Kernel Library (MKL), BigDL enabled deep learning directly within Spark without requiring external frameworks (Intel, 2019). This tight integration offered strong fault tolerance and simplified deployment in JVM-based environments. However, BigDL's custom APIs diverge significantly from mainstream frameworks like TensorFlow and PyTorch, increasing the learning curve and limiting portability of existing models. Furthermore, the lack of GPU acceleration significantly restricts performance for complex networks. While effective for certain enterprise use cases on CPU infrastructure, BigDL remains less suitable for cutting-edge deep learning tasks or flexible experimentation.

D. DeepLearning4J (DL4J)

DeepLearning4J (Gibson Adam *et al.*, 2016) is a Java-based deep learning framework that integrates with Hadoop and Spark and supports distributed training through a parameter server model. Its alignment with the JVM ecosystem makes it suitable for enterprise environments using Java or Scala. However, compared to frameworks like TensorFlow or PyTorch, DL4J has a smaller ecosystem and community. Moreover, similar to other frameworks its parameter server model may introduce overhead in large-scale deployments , and its Java-centric design can limit flexibility in Python-based workflows.

E. Horovod

Horovod (Sergeev and Del Balso, 2018), introduced by Uber, offers an efficient all-reduce-based strategy for distributed training, achieving near-linear scaling across multiple GPUs. Its integration with Spark via Horovod-on-Spark (Horovod, 2019) leverages Spark's barrier execution mode to coordinate training processes with minimal changes to existing TensorFlow or PyTorch code. However, Horovod's reliance on MPI or NCCL introduces operational complexity and demands careful environment configuration, particularly in heterogeneous or CPU-based clusters. It also performs communication outside Spark's ecosystem, limiting visibility and fault tolerance from Spark's perspective. While highly effective in GPU-rich environments, Horovod's added dependencies and setup overhead make it less suitable for lightweight Spark clusters.

F. Spark-tensorflow-distributor

Spark-tensorflow-distributor (TensorFlow, 2021) is a streamlined library that enables distributed TensorFlow training on Spark using its barrier execution mode. Unlike earlier solutions like TFoS, it avoids the need for parameter servers and supports native TensorFlow distribution strategies such as MultiWorkerMirroredStrategy (TensorFlow, 2024), simplifying deployment and code reuse. The library automates critical tasks, including chief/worker designation and inter-node communication, thereby reducing developer effort and facilitating code reuse. Its compatibility with both CPUs and GPUs adds flexibility, and its ability to run minimally modified TensorFlow scripts makes it accessible for teams already using the TensorFlow ecosystem. However, communication overhead during synchronous training remains a challenge especially on CPU-bound clusters. While not as optimized as Horovod for GPU-heavy tasks, it strikes a practical balance between usability, performance, and ecosystem integration—particularly suited to Spark-centric workflows. The overall architecture is shown in Fig. 2, where the Spark driver coordinates distributed TensorFlow workers across executors to perform parallel training.

G. RayDP

RayDP (OAP, 2021) integrates Apache Spark with Ray, a general-purpose distributed computing framework, to support scalable machine learning workloads including distributed TensorFlow training. It enables Spark to preprocess data and Ray to handle model training, facilitating flexible resource sharing between data and training jobs. This hybrid architecture offers greater elasticity and fine-grained control over resource scheduling compared to Spark's native tools. However, managing dual runtimes (Spark and Ray) introduces operational complexity, particularly in deployment and monitoring. Additionally, inter-framework communication and coordination can add latency and increase system overhead. While promising for advanced use cases requiring dynamic resource management, RayDP may be excessive for simpler workflows.

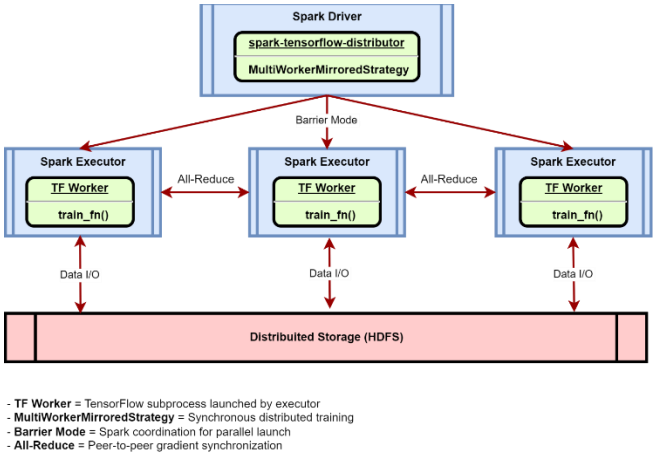

Fig. 2: Architecture of distributed training using Spark TensorFlow Distributor with MultiWorkerMirroredStrategy. The Spark driver uses barrier mode to launch training tasks across executors. Each executor runs a TF worker executing train_fn(), with gradient synchronization via All-Reduce.

III. Methodology

The experimental pipeline is structured to exploit the parallel processing capabilities of Spark for efficient data handling and model training. The overall system architecture and data flow are depicted in Fig. 3, which outlines the sequential stages of the methodology, including data ingestion, preprocessing, training, and evaluation.

A. System Architecture

The experiment was conducted on a single-node Hadoop 3.4 cluster (Ubuntu 20.04) deployed via VirtualBox with 10 CPUs, 10 GB RAM, and 120 GB SSD (no GPU). To simulate a real multi-node distributed environment, Spark 3.4.3 was used in Standalone mode. A Spark worker was created on the same physical machine, allocated 4 GB of memory and 4 CPUs. Inside this worker, two Spark executors were configured, each with 2 GB of memory and 2 CPUs. The single-node limitation means there is no network latency between executors, but there is still inter-process communication overhead. The remaining resources were allocated to the OS, the Jupyter Notebook running the PySpark driver, and the executors' memory overhead.

B. Dataset

CIFAR-10 (Krizhevsky, 2009) consists of 60,000 32×32 RGB images across 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). The dataset is split into 50,000 training images and 10,000 testing images. Despite its modest 170 MB size, it serves as a useful benchmark for evaluating distributed training. Images were provided in binary format and stored on HDFS for efficient parallel processing. A visual overview of randomly sampled images is provided in Fig. 4, highlighting the diversity and resolution of CIFAR-10 images across various classes.

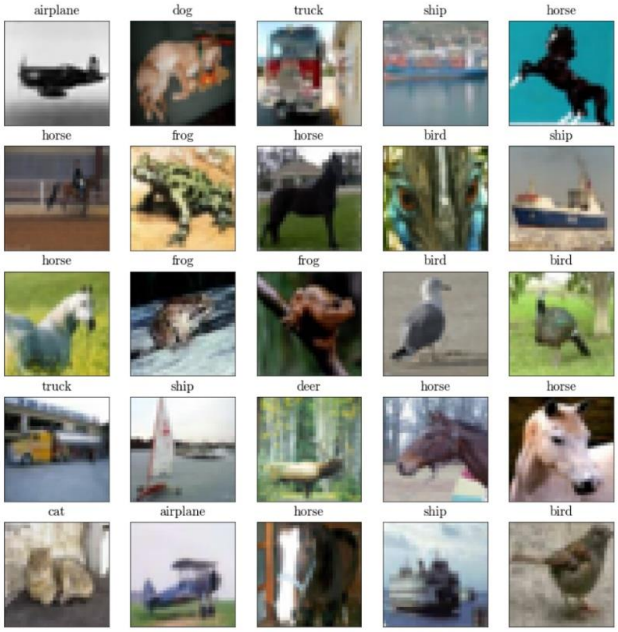

Fig. 4: Sample images from the CIFAR-10 dataset, showcasing the diversity of classes including airplanes, ships, animals, and vehicles. Each image is 32×32 pixels in RGB format. The dataset contains 60,000 such images across 10 distinct classes

C. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) was performed using Apache Spark to evaluate the dataset's integrity and the balance of its classes. Due to memory limitations on the driver node, Out of Memory (OOM) errors were encountered during the analysis. To mitigate this issue, a sampling technique was

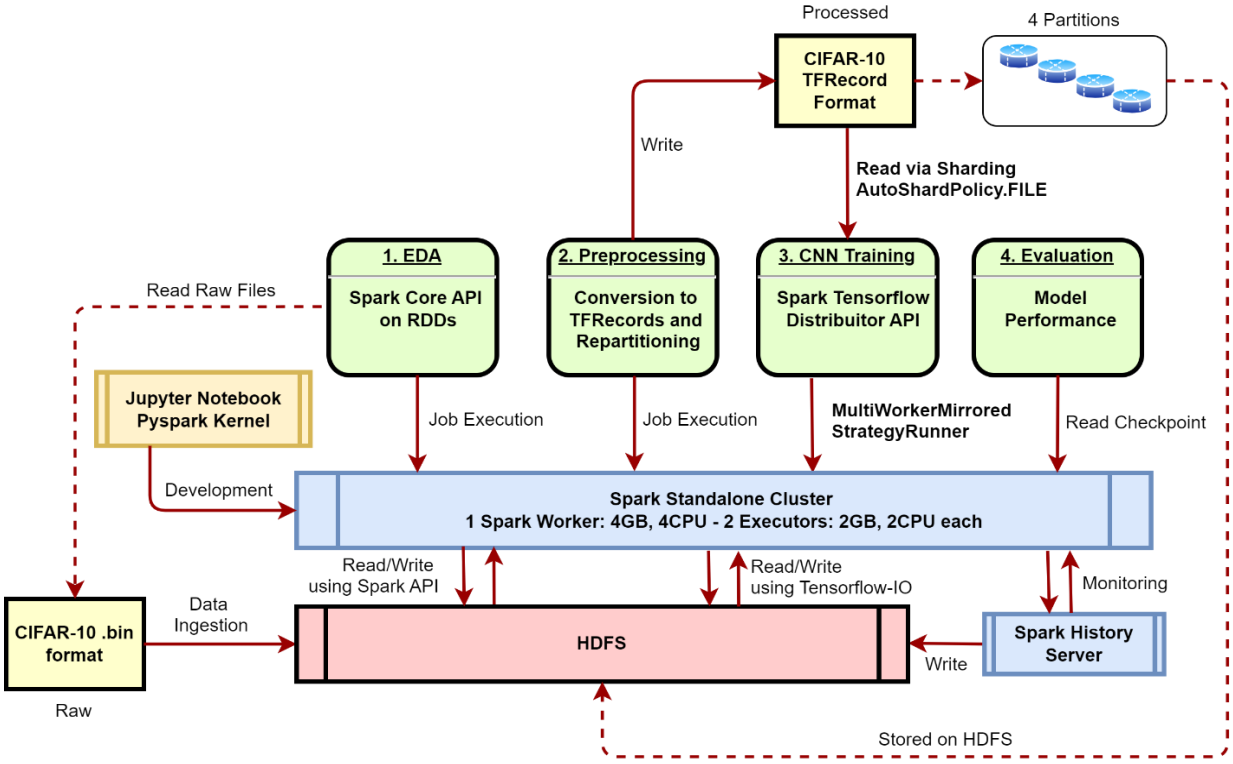

Fig. 3: System architecture and data flow diagram illustrating the end-to-end distributed deep learning pipeline using Apache Spark and HDFS with the CIFAR-10 dataset.

employed to select a representative subset of the data (Pan *et al.*, 2023), ensuring that the statistical properties of the full dataset were preserved while reducing the memory footprint. This approach allowed for efficient analysis without compromising the validity of the results.

Binary parsing was utilized to optimize data loading and enhance the performance of the processing pipeline. The distribution of class labels was analyzed to identify potential imbalances, which could affect model training. Pixel-level statistics were also computed to examine the distribution of features within the images, which is crucial for image classification tasks. A comprehensive breakdown of the EDA, including the sampling process, label distributions, and pixel statistics, is provided in Appendix A.

D. Pre-processing

To emulate large-scale distributed workloads, CIFAR-10 was converted to TFRecord format and stored on HDFS. Data was partitioned into shards for parallel, non-redundant processing by Spark executors. Although TFRecord may introduce I/O overhead when used with small datasets (TensorFlow, 2023), its use was deemed essential for emulating a big data environment.

Standard image normalization was also applied by scaling pixel values from the original range of 0–255 to [0,1], a critical step for accelerating convergence in deep learning models (Krizhevsky, Sutskever and Hinton, 2017). Labels were converted into integer indices to ensure compatibility with the categorical cross-entropy loss function. Data augmentation was not applied but noted for future experiments.

E. CNN Model Architecture

A compact CNN was developed, inspired by VGG (Simonyan and Zisserman, 2014) and baseline CIFAR-10 models (Liu and Deng, 2015). It consists of convolutional, batch normalization, pooling, and dropout layers, followed by a fully connected softmax output layer. The model includes ~592K trainable parameters, balancing complexity and resource constraints. An overview of the model architecture is shown in Fig. 5, and full specifications including activation functions and dropout rates are provided in Appendix B. While models such as ResNet-56 and Wide ResNet have been designed to achieve accuracies exceeding 90% on CIFAR-10 (He *et al.*, 2015; Zagoruyko and Komodakis, 2016), a smaller architecture was intentionally selected to enable reasonable training times on CPU-based resources.

Adam optimizer was employed due to its faster convergence compared to standard Stochastic Gradient Descent (SGD) on smaller models (Kingma and Ba, 2015). In distributed training with multiple workers processing batches in parallel, the effective batch size increases, warranting linear scaling of the learning rate with the number of workers (Goyal *et al.*, 2017). However, for simplicity, a fixed learning rate of 0.001 was used.

Sparse categorical cross-entropy loss was used to handle integer-encoded labels, a standard approach for classification tasks (Goodfellow, Bengio and Courville, 2016). Accuracy, defined as the ratio of correct predictions to total instances, was used as the primary evaluation metric due to its simplicity and interpretability (Hand and Till, 2001).

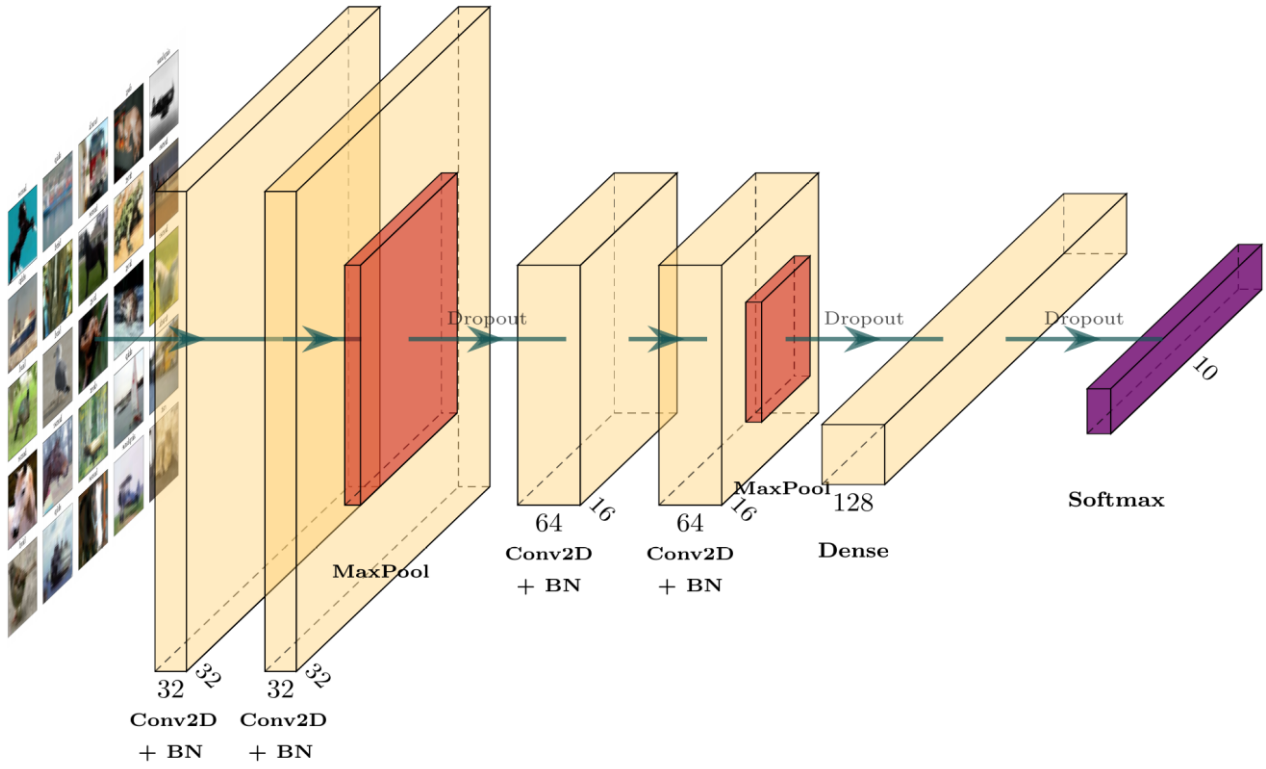

Fig. 5: Visual overview of the proposed CNN architecture for CIFAR-10 classification. The network consists of alternating convolutional and batch normalization layers, interleaved with pooling and dropout layers. Extracted features are flattened and passed through a dense fully-connected layer before final classification into one of 10 categories via a softmax layer. Diagram generated using PlotNeuralNet (Iqbal, 2018).

F. *Distributed Training Procedure*

Distributed training used Spark-tensorflow-distributor with TensorFlow's MultiWorkerMirroredStrategy (TensorFlow, 2024) for synchronous gradient updates. The dataset was sharded using AutoShardPolicy.FILE to ensure each worker processed unique data. A batch size of 16 per worker was selected for memory stability. Training steps were calculated to match full dataset traversal per epoch. Dataset.repeat() ensured uninterrupted data flow. EarlyStopping with a patience of 5 was employed to manage convergence, and checkpoints were saved to HDFS by the chief worker to avoid file conflicts. Fig. 7 illustrates the distributed training job's execution in Spark, highlighting the Directed Acyclic Graph (DAG) structure operating in barrier mode. This mode enforces synchronization among all workers at each step, aligning with TensorFlow's requirement for consistent gradient updates across workers.

G. *Monitoring*

Training jobs were monitored through the Spark History Server (Fig. 8), providing real-time insights into job status, resource utilization, and execution metrics. Event logs generated during training were stored in the Hadoop Distributed File System (HDFS) for subsequent analysis, enabling the identification of performance bottlenecks and resource inefficiencies. However, while the Spark History Server effectively tracks Spark job execution, it lacks visibility into the internal processes of TensorFlow. To capture detailed training progress, model metrics, and TensorFlow-specific events, the use of TensorBoard would be necessary. For the sake of simplicity and to maintain focus on distributed training aspects, TensorBoard was not employed in this study.

Fig. 7: Spark DAG visualization for the distributed training stage executed in barrier mode using Spark TensorFlow Distributor. A ParallelCollectionRDD is created and processed in a synchronous manner via a PythonRDD that invokes the training function under TensorFlow's MultiWorkerMirroredStrategy.

IV. Results and Discussion

The primary objective of this study is to evaluate whether distributed training using Apache Spark and TensorFlow could achieve accuracy and convergence comparable to single-worker training. Experiments conducted on the CIFAR-10 dataset with 1-slot (single executor) and 2-slot (two executors) configurations demonstrated stable convergence, achieving approximately 70% test accuracy. Training was conducted for up to 50 epochs, employing early stopping with a patience of 5 epochs based on validation loss. If no improvement in validation loss was observed over 5 consecutive epochs, training was halted, and metrics from the best-performing epoch were recorded to mitigate overfitting.

A. *Training Accuracy and Convergence*

As illustrated in Fig. 9 the 1-slot configuration exhibited a rapid improvement in accuracy during the initial 10 epochs, stabilizing between approximately 0.70 and 0.72 by epoch 15. Conversely, the 2-slot setup converged more slowly,

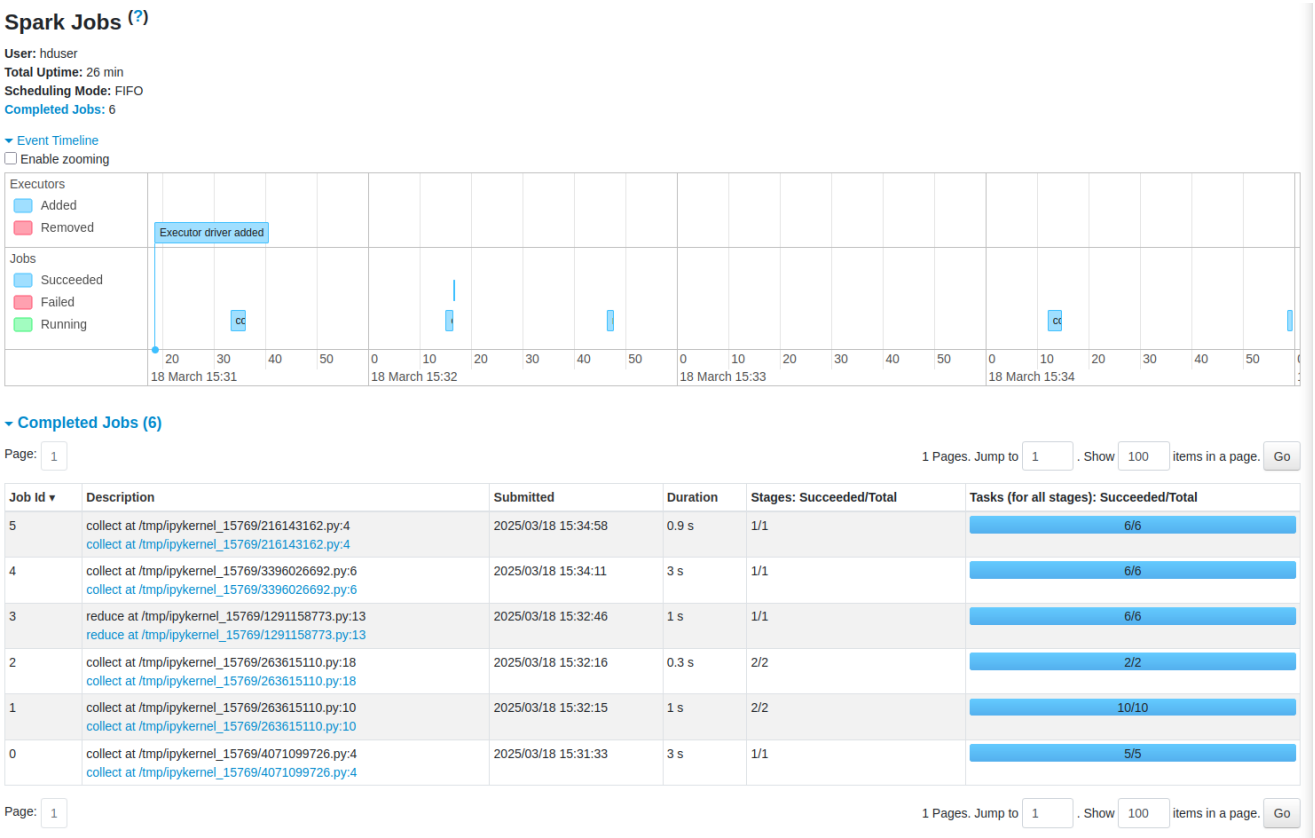

Fig. 6: Spark History Server interface showing completed Spark jobs. Each job includes detailed metadata such as the job ID, execution duration, stage and task completion, and the script location from which it was invoked.

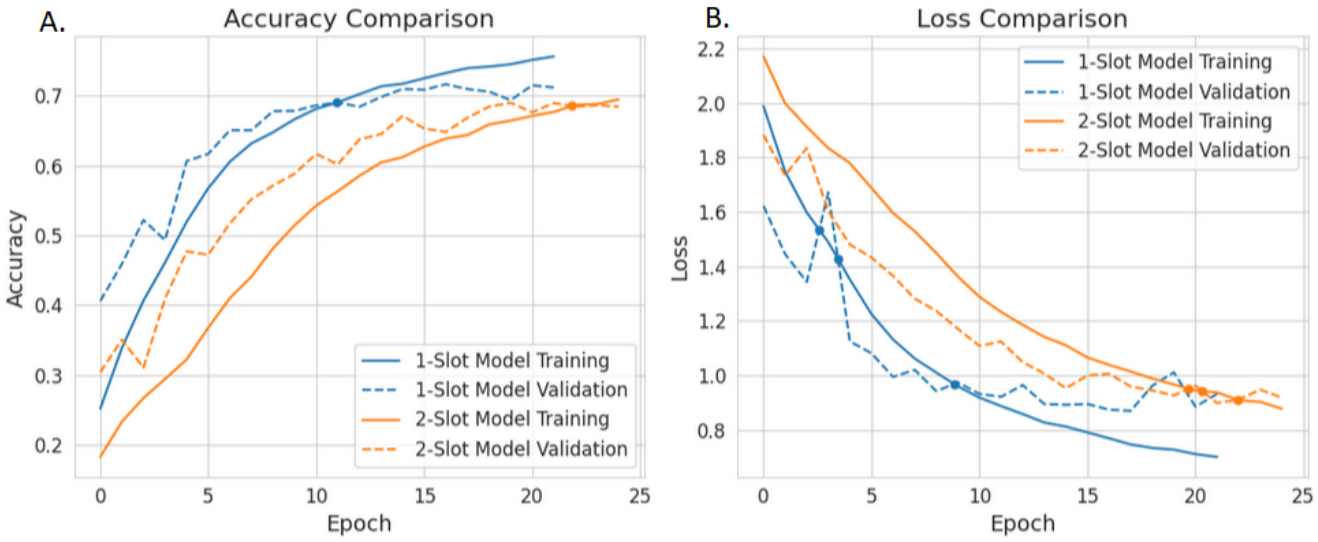

Fig. 8: (A) Accuracy comparison between the single-slot and two-slot models. The single-slot model reaches higher accuracy earlier and stabilizes slightly above the two-slot configuration. (B) Loss comparison shows that while both models exhibit decreasing training and validation loss, the two-slot model converges more gradually, likely due to synchronization overhead during distributed training.

achieving stability between 0.66 and 0.68 accuracy around epoch 20. Similar patterns are evident in the loss curves, where training and validation losses steadily declined; however, the loss reduction was more gradual for the 2-slot model, likely due to overhead introduced by gradient synchronization and dataset partitioning. Ultimately, the 2-slot configuration reached approximately 0.68 accuracy with a validation loss of 0.96, whereas the 1-slot model attained approximately 0.71 accuracy and a lower validation loss of 0.87. Table II summarizes these findings, highlighting a modest performance difference of 0.0332 in accuracy and 0.0921 in test loss between the two configurations.

B. *Training Time and Efficiency*

Distributed training achieved a runtime reduction of approximately 14%, decreasing from 56 minutes and 39 seconds (1-slot) to 48 minutes and 35 seconds (2-slot), as shown in Table II. However, the speedup was noticeably sublinear; in an ideal scenario, doubling the number of executors would result in a halved training time. In practice, performance gains were constrained by various factors, including synchronization overhead, shared hardware resources, and latency introduced by TensorFlow's all-reduce operations. Additionally, scalability was further impeded by thread contention, memory bandwidth saturation, and data sharding overhead—issues particularly pronounced within CPU-bound, single-node environments. These challenges underscore the complexities inherent in optimizing distributed deep learning systems.

TABLE II. PERFORMANCE COMPARISON OF 1-SLOT AND 2-SLOT TRAINING CONFIGURATIONS

Metric	**Performance**		
	1-Slot Model	***2-Slot Model***	***Difference***
Test Loss	0.869453	0.961556	0.0921
Test Accuracy	0.7093	0.676082	-0.0332
Training Time	56m 39.43s	48m 35.22s	-8.07 minutes

C. *MNIST Baseline*

Before transitioning to CIFAR, the infrastructure was validated using a simple CNN on the MNIST dataset (LeCun and Cortes, 2005), a simpler classification task with grayscale images. The model achieved over 98% test accuracy after a single epoch, indicating minimal training complexity. Both the 1-slot and 2-slot configurations performed similarly, with no significant differences in accuracy or runtime observed. While MNIST confirmed the stability of the distributed setup, its simplicity underscored the need for a more complex model and dataset to properly evaluate the benefits of distributed training, prompting subsequent experiments on CIFAR.

D. *Limitations and Challenges*

1) System Constraints: The single-node environment allocated 2 GB RAM and 2 CPU cores per executor, which imposed significant constraints on TensorFlow operations. Under the 2-slot configuration, system instability and occasional crashes were observed due to concurrent executor activity. OS-level monitoring showed sustained near-saturation of memory and CPU resources, indicating the setup operated at its limits. These findings underscore the need for careful tuning of executor configurations and sufficient resource provisioning when scaling distributed training workloads.

2) Model Quality Considerations: Gradient staleness, a common issue in distributed training caused by synchronization delays, can lead to outdated model weights (Zinkevich *et al.*, 2010).To address this, a fully synchronous training strategy was employed. However, the slight drop in accuracy in the 2-slot configuration may hint at early signs of staleness. As executor count increases, synchronization becomes more critical. Despite this, convergence patterns and test loss indicate minimal impact in this setup. Further evaluation with more executors is needed to fully assess the effect.

3) Checkpointing in Distributed Contexts: Model checkpointing was managed by assigning worker 0 as the chief, responsible for saving the final model to HDFS. This

avoided race conditions and file conflicts that could occur if all workers attempted to write concurrently. Though a minor limitation, it underscores the need for improved tooling for coordinated model saving in distributed settings. Integrating MLflow could enhance this process by providing structured model versioning, tracking, and artifact management (Zaharia *et al.*, 2018).

E. Research Gaps and Future Improvements

1) Fault Tolerance and Elasticity: While Spark offers strong fault tolerance for data processing, its current barrier execution mode lacks resilience during distributed training, any single worker failure results in complete job termination. Enhancing robustness for long-running tasks may require periodic checkpointing and finer-grained recovery mechanisms. Elastic training, which allows dynamic addition or removal of workers mid-job (as supported in frameworks like PyTorch and Kubernetes), presents a promising direction for improved resource flexibility (PyTorch, 2024). Although not yet supported in Spark's native tooling, this represents a valuable opportunity for future development in dynamic resource management.

2) Integration with Broader Hadoop Ecosystem: Beyond Spark, the Hadoop ecosystem includes tools like Hive and HBase that support diverse data services. Using Spark connectors, models could be trained directly on data from a Hive warehouse or a NoSQL store like HBase, enabling seamless integration of analytics and machine learning pipelines. Although not explored in this study, such integration offers a promising path for production workflows, enhancing efficiency and scalability by unifying data processing and model training.

3) Advanced Analytics Improvements: The model achieves approximately 70% accuracy on CIFAR-10, a reasonable result given the study's intentional constraints. Techniques such as data augmentation, learning rate scheduling, and advanced architectures like ResNet-50 (He et al., 2015) were excluded due to limited resources. Future work could explore training near state-of-the-art models, which can surpass 95% accuracy with augmentation, on GPU-enabled Spark clusters. This would not only enhance model performance but also serve as a benchmark for evaluating Spark's ability to scale modern deep learning workloads in distributed, big data environments.

4) Distributed Training Monitoring: Although not the primary focus of this work, effective monitoring is essential for distributed training workflows. The Spark History Server was used to track resource usage and job execution, but its visibility is limited since Spark primarily orchestrates TensorFlow jobs, with training logic running in barrier mode. This execution pattern obscures internal model operations from Spark's perspective. For model-level insights, tools like TensorBoard, configured to read logs from a shared HDFS directory, remain necessary. A unified monitoring solution that combines system-level resource tracking with deep learning metrics would greatly enhance observability and debugging in such distributed workflows.

F. Comparison to State-of-the-Art Methods

In this study Spark-tensorflow-distributor (TensorFlow, 2021) provided a straightforward approach to distributed training, requiring minimal code changes while supporting best practices like batch size and learning rate adjustments. Its architecture effectively combines the strengths of Apache Spark (Zaharia *et al.*, 2016) and TensorFlow (Abadi *et al.*, 2016), addressing limitations seen in earlier integration frameworks.

For example, CaffeOnSpark (Yahoo, 2016), though native Spark integration, is built on the Caffe framework, which lacks support for modern distributed training strategies and TensorFlow's broader ecosystem. TensorFlowOnSpark (Yahoo, 2017) bridges Spark and TensorFlow through a custom coordination layer, but often requires additional configuration, such as environment variables, task orchestration, and I/O management, which complicates deployment, especially in large-scale environments. Other frameworks like BigDL (Dai *et al.*, 2019) and DL4J (Gibson Adam *et al.*, 2016) offer high-performance deep learning with native Java Virtual Machine (JVM) and Spark integration. However, they rely on custom APIs and model definitions that diverge from standard TensorFlow interfaces, making them less suitable for workflows dependent on Python-based tooling or existing TensorFlow models. Similarly, Horovod-on-Spark (Horovod, 2019) enables efficient communication via ring-allreduce and supports large-scale training, but introduces extra dependencies (e.g., MPI, NCCL) and requires careful environment configuration, increasing operational overhead. Emerging solutions like RayDP (OAP, 2021) integrate Spark's data processing with Ray's flexible training capabilities, but managing dual runtimes and resource orchestration across systems adds complexity.

Recent developments such as TorchDistributor in Spark 3.4 (Apache Spark, 2023) highlight a shift toward native support for distributed deep learning within Spark, focused on PyTorch. A comparable solution for TensorFlow would further streamline integration, reduce complexity, and promote adoption in enterprise-scale training pipelines.

V. CONCLUSION

This study demonstrated that distributed training of convolutional neural networks (CNNs) using Apache Spark and TensorFlow, facilitated by Spark-tensorflow-distributor, can reduce training time with only a minimal loss in accuracy. Experimental results demonstrated that distributed training (2 executors) reduced training time by approximately 14%, with only a marginal decrease in classification accuracy (67% vs. 71%) compared to a single-node configuration.

These results highlight the practical viability of combining deep learning workflows with scalable data processing platforms using open-source tools. The integration minimizes data movement, promotes modularity, and simplifies deployment in real-world data analytics pipelines. Although limited by system resources and synchronization overhead, the architecture presented here offers an accessible path toward scalable, distributed deep learning. Future research may investigate the generalization of this method to larger datasets, more complex models, and production-scale distributed environments.

APPENDIX A. EDA

Exploratory Data Analysis (EDA) is conducted to assess the structure and quality of the CIFAR-10 dataset before applying distributed deep learning. Raw binary files are inspected, samples visualized, label distribution analyzed, and pixel statistics assessed to identify issues impacting model performance. Data diversity is examined through pixel range distribution to ensure readiness for robust model training.

A.1. Binary File Inspection

The CIFAR-10 dataset (Krizhevsky, Nair and Hinton, 2009) is stored in a binary format in which each record consists of 3073 bytes, 1 byte representing the label (a class ID ranging from 0 to 9), followed by 3072 bytes of pixel data (1024 bytes for each RGB channel). The dataset is loaded using Spark's binaryFiles() function, through which each file is read as a (filename, bytes) pair. A custom parsing function was implemented to split each 3073-byte record into its respective label and pixel components, and to convert the raw bytes into numerical arrays.

A.2. Visual Inspection

Visual inspection plays a key role in ensuring data integrity, especially in distributed systems where operations occur in parallel and errors may go unnoticed without explicit checks. Manual validation methods, such as reviewing samples of parsed data, help confirm correctness and detect potential anomalies, which is essential in maintaining reliability during complex distributed processing tasks (Beschastnikh *et al.*, 2016). An example of such manual inspection is shown in Fig. 4, where randomly selected images from the CIFAR-10 dataset are displayed with their corresponding labels.

A.3. Label Distribution

Ensuring a balanced class distribution is important for unbiased model training and evaluation, as class imbalance can lead to poor generalization and reduced performance (Chawla *et al.*, 2011). Since CIFAR-10 is designed to be balanced, verifying the label distribution provides a quick check that no labels are missing or misclassified during parsing. Fig. 11 confirms that both the training and test datasets contain an even number of samples per class.

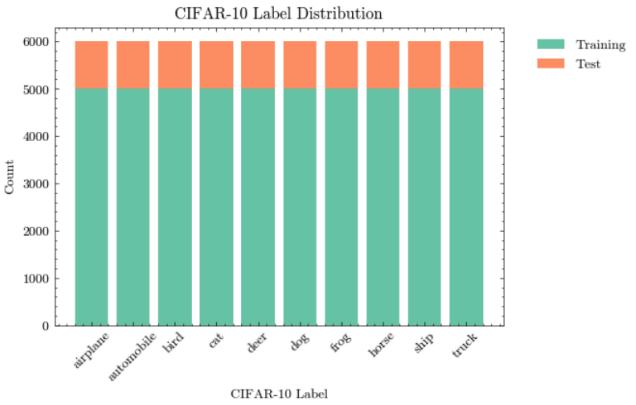

Fig. 10: Label distribution of the CIFAR-10 dataset, confirming balanced class representation across training and test subsets. Each of the 10 classes contains 5,000 training and 1,000 test images, ensuring unbiased model evaluation.

A.4. Brightness and Contrast Analysis

Analyzing pixel intensity statistics, specifically the mean (brightness) and standard deviation (contrast), provides valuable insight into image quality and informs necessary preprocessing steps. Since CIFAR-10 pixel values range from 0 to 255, these metrics help detect potential anomalies such as overly dark, bright, or low-contrast images that could negatively impact model performance. Normalization, a critical preprocessing technique, mitigates such issues by standardizing input distributions, thereby improving training stability and convergence (Ioffe and Szegedy, 2015).

Across the dataset, the average pixel value is approximately 120.84, suggesting a moderately bright distribution. This indicates that the dataset comprises a diverse range of lighting conditions without extreme bias toward dark or bright imagery. The Brightness Distribution (see Fig. 10A.) shows that most images have a mean intensity between 120 and 140, with a relatively symmetric distribution. This implies that lighting conditions are generally balanced, although a small number of images fall at the lower and upper extremes of brightness. The Contrast Distribution (see Fig. 10B.) reveals that the majority of images have a standard deviation of 40–60, indicating sufficient contrast for object

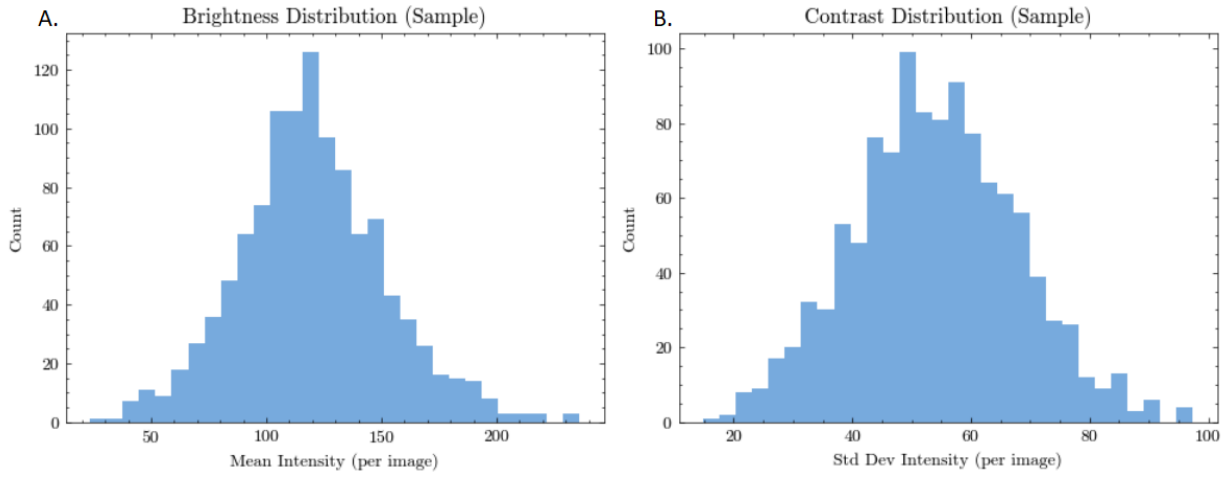

Fig. 9: Pixel intensity analysis of the CIFAR-10 dataset. (A) Brightness distribution based on mean pixel intensity per image. (B) Contrast distribution based on standard deviation of pixel intensity per image. The histograms confirm well-balanced lighting and adequate contrast in most samples, with a few outliers.

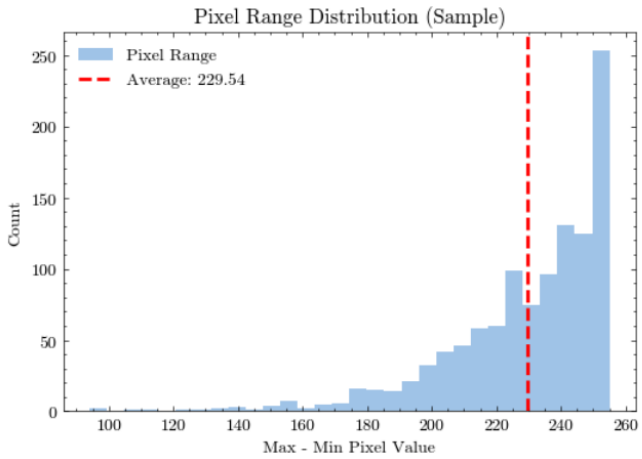

Fig. 11: Pixel range distribution across a sample of CIFAR-10 images. Most images exhibit a full-range spread close to 255, indicating diverse intensity variation. The red dashed line marks the average pixel range of approximately 229.54.

differentiation. However, a minor subset of images falls below a standard deviation of 20, which may correspond to low-detail or blurred samples. Identifying and potentially enhancing or excluding these low-contrast images could improve overall training effectiveness.

A.5. Data Diversity

Assessing image diversity through metrics like pixel value range and intra-image variance is crucial for detecting data quality concerns, such as redundancy or insufficient variation, which could cause model overfitting. To mitigate these issues, data augmentation strategies, including cropping, flipping, and adjusting brightness, are frequently employed to enhance data variability and model generalization capabilities (Shorten and Khoshgoftaar, 2019).

The Pixel Range Distribution, shown in Fig. 12 measures the spread between the minimum and maximum pixel values within each image. A high range indicates that an image utilizes the full intensity scale (0 to 255), capturing a wide spectrum of brightness and contrast. As illustrated in the figure, the majority of CIFAR-10 images have a pixel range close to 255, suggesting strong diversity in lighting and texture. However, a small subset of images exhibits a narrower range (below 200), which may indicate lower contrast or dominance of mid-tone pixel values. Identifying such cases can help guide targeted preprocessing or augmentation strategies to enhance dataset quality.

APPENDIX B. CNN MODEL ARCHITECTURE

The Convolutional Neural Network (CNN) designed for CIFAR-10 classification is built from scratch (see Fig. 5), inspired by proven architectures such as VGG and other baseline models (He *et al.*, 2015; Zagoruyko and Komodakis, 2016). Below are the key components of the network.

B.1. Convolutional Layers

The model uses small 3×3 filters in the convolutional layers, as recommended by the VGG network. Stacking multiple 3×3 convolutional layers allows the network to capture complex features while maintaining fewer parameters compared to using a single larger filter. For example, a sequence of two 3×3 convolutional layers effectively achieve a receptive field of 5×5, and three layers can represent a 7×7 receptive field. The architecture consists of two convolutional layers per block, followed by a pooling layer. The first convolutional block uses 32 filters per layer, and the second block uses 64 filters. All convolutional layers use ReLU activation (Nair and Hinton, 2010) which is widely used for its simplicity and effectiveness in deep networks.

B.2. Pooling Layers

After each pair of convolutional layers, Max Pooling with a 2×2 pool size is applied to down-sample the feature maps. Through this operation, translational invariance is introduced and the spatial dimensions are reduced, an important consideration given the modest image size of 32×32 pixels. A 2×2 pooling configuration is employed, resulting in the halving of each spatial dimension, consistent with common CNN practices (Simonyan and Zisserman, 2014). Following two successive pooling operations, the spatial size is reduced to 8×8, which is considered manageable for processing by the fully connected layer.

B.3. Batch Normalization

Batch Normalization (BN) is applied after each convolutional layer. BN normalizes the inputs of each layer, helping to stabilize training and allow the use of higher learning rates (Ioffe and Szegedy, 2015). Although the original VGG network did not use BN, modern CNN architectures such as ResNet heavily incorporate it due to its advantages in reducing internal covariate shift and improving convergence (He *et al.*, 2015). Including BN helps mitigate distribution shift when training in parallel on multiple executors.

B.4. Fully Connected (Dense) Layers

After flattening the pooled feature maps, a dense layer with 128 units and ReLU activation (Nair and Hinton, 2010) is applied. This layer learns to combine the extracted features before the final output. The choice of 128 units reflects a balance between model capacity and computational efficiency, supported by prior work on small CNNs for CIFAR-10 (Krizhevsky, 2009).

B.5. Output Layer

The output layer is a dense layer with 10 units (one per CIFAR-10 class), using a softmax activation function to generate a probability distribution over the class labels. Softmax is commonly used in multi-class classification problems for its probabilistic interpretation (Bridle, 1990).

B.6. Regularization (Dropout)

Dropout layers are inserted to combat overfitting, which can be significant with a small dataset like CIFAR-10. A dropout rate of 0.25 is used after each pooling layer (on the convolutional feature maps), and 0.5 on the dense layer. The use of a 0.5 dropout rate in fully connected layers follows early empirical findings demonstrating its effectiveness in regularizing neural networks and reducing co-adaptation of neurons (Srivastava *et al.*, 2014). In distributed training, dropout offers the added benefit that each executor randomly drops different neurons (due to separate random seeds), potentially improving the robustness of the aggregated model.

REFERENCES

Abadi, M. *et al.* (2016) 'TensorFlow: A system for large-scale machine learning'. arXiv. doi: 10.48550/arXiv.1605.08695.

Apache Spark (2018) *Spark Release 2.4.0*. Available at: https://spark.apache.org/releases/spark-release-2-4-0.html (Accessed: 19 March 2025).

Apache Spark (2023) *Spark Release 3.4.0*. Available at: https://spark.apache.org/releases/spark-release-3-4-0.html (Accessed: 19 March 2025).

Beschastnikh, I. *et al.* (2016) 'Debugging distributed systems', *Communications of the ACM*. doi: 10.1145/2909480.

Bridle, J. S. (1990) 'Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition', *NATO Neurocomputing*, pp. 227–236. doi: 10.1007/978-3-642-76153-9_28.

Chawla, N. V. *et al.* (2011) 'SMOTE: Synthetic Minority Over-sampling Technique', *arXiv: Artificial Intelligence*. doi: 10.1613/jair.953.

Chen, X. and Lin, X. (2014) 'Big Data Deep Learning: Challenges and Perspectives', *IEEE Access*, 2, pp. 514–525. doi: 10.1109/access.2014.2325029.

Dai, J. *et al.* (2019) 'BigDL: A Distributed Deep Learning Framework for Big Data', *ACM Symposium on Cloud Computing*, pp. 50–60. doi: 10.1145/3357223.3362707.

Dean, J. *et al.* (2012) 'Large scale distributed deep networks', in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. Red Hook, NY, USA: Curran Associates Inc. (NIPS'12), pp. 1223–1231.

Feng, A. *et al.* (2016) *CaffeOnSpark Open Sourced for Distributed Deep Learning on Big Data Clusters, Yahoo Developer Network*. Available at: https://developer.yahoo.com/blogs/139916563586/ (Accessed: 19 March 2025).

Gibson Adam *et al.* (2016) 'Deeplearning4j: Distributed, open-source deep learning for Java and Scala on Hadoop and Spark'. doi: 10.6084/m9.figshare.3362644.v2.

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. MIT Press.

Goyal, P. *et al.* (2017) 'Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour', *arXiv: Computer Vision and Pattern Recognition*. doi: 10.48550/arXiv.1706.02677.

Hand, D. J. and Till, R. (2001) 'A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems', *Machine Learning*, 45(2), pp. 171–186. doi: 10.1023/a:1010920819831.

He, K. *et al.* (2015) 'Deep Residual Learning for Image Recognition', *arXiv: Computer Vision and Pattern Recognition*. doi: 10.1109/cvpr.2016.90.

Horovod (2019) *Horovod on Spark*. Available at: https://horovod.readthedocs.io/en/stable/spark_include.html (Accessed: 20 March 2025).

Intel (2019) *BigDL*. Available at: https://bigdl-project.github.io/0.7.0/ (Accessed: 20 March 2025).

Ioffe, S. and Szegedy, C. (2015) 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift', *arXiv: Learning*. doi: 10.48550/arXiv.1502.03167.

Jordan, M. I. and Mitchell, T. M. (2015) 'Machine learning: Trends, perspectives, and prospects', *Science*, 349(6245), pp. 255–260. doi: 10.1126/science.aaa8415.

Kingma, D. P. and Ba, J. (2015) 'Adam: A Method for Stochastic Optimization', *International Conference on Learning Representations*. doi: 10.48550/arXiv.1412.6980.

Krizhevsky, A. (2009) 'Learning Multiple Layers of Features from Tiny Images'. Available at: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Krizhevsky, A., Nair, V. and Hinton, G. (2009) *CIFAR-10 and CIFAR-100 datasets*. Available at: https://www.cs.toronto.edu/~kriz/cifar.html (Accessed: 21 March 2025).

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2017) 'ImageNet classification with deep convolutional neural networks', *Communications of The ACM*, 60(6), pp. 84–90. doi: 10.1145/3065386.

LeCun, Y., Bengio, Y. and Hinton, G. E. (2015) 'Deep learning', *Nature*, 521(7553), pp. 436–444. doi: 10.1038/nature14539.

LeCun, Y. and Cortes, C. (2005) 'The mnist database of handwritten digits', in. Available at: https://www.semanticscholar.org/paper/The-mnist-database-of-handwritten-digits-LeCun-Cortes/dc52d1ede1b90bf9d296bc5b34c9310b7eaa99a2 (Accessed: 24 March 2025).

Li, M. *et al.* (2014) 'Scaling distributed machine learning with the parameter server', *USENIX Symposium on Operating Systems Design and Implementation*, pp. 583–598. doi: 10.5555/2685048.2685095.

Liu, S. and Deng, W. (2015) 'Very deep convolutional neural network based image classification using small training sample size', *Asian Conference on Pattern*

Recognition, pp. 730–734. doi: 10.1109/acpr.2015.7486599.

Nair, V. and Hinton, G. E. (2010) 'Rectified Linear Units Improve Restricted Boltzmann Machines', *International Conference on Machine Learning*, pp. 807–814. doi: 10.5555/3104322.3104425.

OAP (2021) 'RayDP'. Available at: https://github.com/oap-project/raydp (Accessed: 19 March 2025).

Pan, R. *et al.* (2023) 'A Sequential Addressing Subsampling Method for Massive Data Analysis Under Memory Constraint', *IEEE Transactions on Knowledge and Data Engineering*, 35(9), pp. 9502–9513. doi: 10.1109/tkde.2023.3241075.

Paszke, A. *et al.* (2019) 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. doi: 10.48550/arxiv.1912.01703.

PyTorch (2024) *Torch Distributed Elastic*. Available at: https://pytorch.org/docs/stable/distributed.elastic.html (Accessed: 23 March 2025).

Sergeev, A. and Del Balso, M. (2018) 'Horovod: fast and easy distributed deep learning in TensorFlow', *arXiv (Cornell University)*. doi: 10.48550/arxiv.1802.05799.

Sewal, P. and Singh, H. (2021) 'A Critical Analysis of Apache Hadoop and Spark for Big Data Processing', *2021 6th International Conference on Signal Processing, Computing and Control (ISPCC)*. doi: 10.1109/ispcc53510.2021.9609518.

Shorten, C. and Khoshgoftaar, T. M. (2019) 'A survey on Image Data Augmentation for Deep Learning', *Journal of Big Data*, 6(1), pp. 1–48. doi: 10.1186/s40537-019-0197-0.

Simonyan, K. and Zisserman, A. (2014) 'Very Deep Convolutional Networks for Large-Scale Image Recognition', *International Conference on Learning Representations*. doi: 10.48550/arXiv.1409.1556.

Srivastava, N. *et al.* (2014) 'Dropout: a simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research*, 15(1), pp. 1929–1958. doi: 10.5555/2627435.2670313.

TensorFlow (2021) 'spark-tensorflow-distributor'. Available at: https://github.com/tensorflow/ecosystem/tree/master/spark/spark-tensorflow-distributor (Accessed: 19 March 2025).

TensorFlow (2024) *API Documentation | TensorFlow v2.16.1*. Available at: https://www.tensorflow.org/api_docs (Accessed: 22 March 2025).

Yahoo (2016) 'CaffeOnSpark'. Yahoo. Available at: https://github.com/yahoo/CaffeOnSpark (Accessed: 19 March 2025).

Yahoo (2017) 'TensorFlowOnSpark'. Yahoo. Available at: https://github.com/yahoo/TensorFlowOnSpark (Accessed: 19 March 2025).

Yang, L. *et al.* (2017) *Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters, Yahoo Developer Network*. Available at: https://developer.yahoo.com/blogs/157196317141/ (Accessed: 20 March 2025).

Zagoruyko, S. and Komodakis, N. (2016) 'Wide Residual Networks', *British Machine Vision Conference*. doi: 10.48550/arXiv.1605.07146.

Zaharia, M. *et al.* (2010) 'Spark: cluster computing with working sets', *USENIX Workshop on Hot Topics in Cloud Computing*, p. 10. Available at: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.pdf.

Zaharia, M. *et al.* (2016) 'Apache Spark: a unified engine for big data processing', *Communications of The ACM*, 59(11), pp. 56–65. doi: 10.1145/2934664.

Zinkevich, M. A. *et al.* (2010) 'Parallelized stochastic gradient descent', in *Proceedings of the 24th International Conference on Neural Information Processing Systems - Volume 2*. Red Hook, NY, USA: Curran Associates Inc. (NIPS'10), pp. 2595–2603. doi: 10.5555/2997046.2997185.