Scalable Deep Learning with Spark: A System for Large-Scale Data Processing and Model Training

December 20, 2024

***Abstract*—This paper proposes a distributed data loading method based on the Spark and PyTorch deep learning frameworks, aiming to address the memory bottlenecks and efficiency issues encountered by traditional single-machine data loading when handling large-scale datasets. We have designed and implemented a custom Dataset class that leverages Spark's distributed computing capabilities, allowing models to load data on demand and thereby enhancing the efficiency and scalability of the training process. Experimental validation on both real-world and simulated datasets demonstrates that our approach outperforms traditional models in predictive performance and efficiently handles large-scale datasets, confirming its effectiveness and commercial scalability in practical applications.**

***Index Terms*—Deep Learning, Training, Large-Scale Datasets**

I. Introduction

A. The Problem

In large-scale deep learning modeling, data loading and processing have always been central challenges. As data scales grow rapidly, traditional single-machine training models begin to expose bottlenecks: memory limitations prevent the direct loading of the entire dataset, and the efficiency of single-threaded preprocessing fails to meet the demands of model training. These issues become particularly pronounced in distributed cluster environments. How to dynamically process and load large-scale data stored in distributed storage while maintaining training efficiency is an urgent problem to address.

Existing solutions usually have shortcomings in two dimensions. First, optimization solutions for data loading mostly remain within the single-machine domain, such as improvements based on multi-threading or caching technologies, which cannot be extended to distributed clusters. Second, there is a lack of deep integration between deep learning frameworks and big data processing frameworks, which causes data partitioning, transmission, and loading steps to become performance bottlenecks. This disconnect forces users to develop and debug complex intermediate toolchains when deploying distributed training, adding to system complexity and development costs.

This paper focuses on the core problem of efficiently integrating data loading and model training. In this context, we propose a distributed data processing method based on Spark, which is seamlessly embedded into the PyTorch deep learning framework, enabling dynamic and efficient data loading and training integration. This integration not only solves the memory limitation issue of single machines but also significantly improves data processing speed in distributed environments. Compared to existing solutions, our design avoids interference between data loading and training processes, providing reliable support for deep learning applications in ultra-large-scale data scenarios.

B. Our Approaches

In this paper, we propose a novel distributed data loading method by combining Spark's distributed computing capabilities with the PyTorch deep learning framework. Our design implements a Spark-based custom Dataset class, allowing deep learning models to dynamically load data from the distributed cluster on demand, effectively addressing the memory bottlenecks and efficiency issues associated with traditional single-machine data loading.

C. Our Results

Experiments show that the proposed method performs excellently on both real and simulated datasets. Compared to classical models, our GAT model significantly outperforms the baseline methods in prediction performance while efficiently handling ultra-large-scale data. Furthermore, the system's distributed architecture supports efficient scaling in a multi-node environment, validating its practicality in commercial-grade data scenarios.

II. Related Work

This section reviews related work in the field, with a focus on the integration of distributed data processing and deep learning model training, Spark-based data processing frameworks, and the application of Graph Neural Networks (GNNs) in large-scale data analysis.

A. Distributed Data Processing and Deep Learning Training

As the size of datasets continues to grow, traditional single-machine training methods can no longer meet the demand. Many researchers have begun exploring the application of distributed computing frameworks in deep learning training. Especially in the context of big data, the integration of distributed data processing techniques with deep learning training has become an important research direction. Several works have adopted frameworks such as Apache Spark, TensorFlow, and PyTorch to decouple data processing from model training and leverage distributed computing resources to improve training efficiency. For example, Zaharia et al. (2016) proposed Apache Spark, which provides an efficient and scalable distributed computing engine widely used in large-scale data processing[1]. Additionally, Gupta et al. (2020) introduced a method for combining Spark with deep learning by integrating the data processing process with TensorFlow

model training, thereby enhancing training performance on large-scale datasets[2].

Similar to these studies, this paper successfully integrates the Spark framework with PyTorch's DataLoader, enabling distributed training on large-scale datasets, solving the problem of data that cannot be fully loaded into single-machine memory, while maintaining an efficient training process.

B. Application of Spark Framework in Deep Learning

In the fields of big data analysis and machine learning, Apache Spark is widely used for distributed data processing and training tasks. Spark supports parallel computation of large-scale data and provides efficient memory computing capabilities through RDDs (Resilient Distributed Datasets). Especially through Spark MLlib, many studies have demonstrated its advantages in distributed machine learning. For example, Dai et al. (2019) proposed a method of combining Spark with deep learning algorithms, which improved the computation efficiency of large-scale datasets using RDDs and MLlib[3].

Although Spark has been widely researched for its applications in distributed data processing and machine learning, how to efficiently integrate Spark with deep learning frameworks like PyTorch for large-scale distributed model training remains an open research question. This paper provides a new solution for such tasks by implementing the SparkDataset module, which seamlessly integrates data from the Spark cluster with PyTorch's DataLoader.

C. Graph Neural Networks and Large-Scale Data Analysis

Graph Neural Networks (GNNs) have recently become an important research direction in deep learning, particularly excelling in handling structured and relational data. Many application scenarios, such as social network analysis, recommendation systems, and chemical molecular modeling, can achieve significant performance improvements through GNNs. As a type of GNN, Graph Attention Networks (GAT) enhance the influence of adjacent nodes in a graph by using an attention mechanism between nodes, further improving the model's performance on graph-structured data.

In large-scale data analysis, one of the challenges faced by GNNs is how to efficiently process large-scale graph data. Some existing research has combined GNNs with distributed computing frameworks to improve training efficiency. For example, the GraphSAGE method proposed by Hamilton et al. (2018) reduces the computational pressure caused by graph scale through local aggregation strategies[4]. Similarly, this paper integrates the GAT model with the data processing capabilities of the Spark cluster, enabling effective GNN training even on extremely large datasets.

D. System and User Interface Development

To further enhance the system's usability, many research efforts have focused on the development and optimization of system interfaces. Web interfaces and graphical user interfaces (GUIs) provide intuitive interaction methods for the deep learning model training process, allowing non-expert users to efficiently monitor and control the training process. Similar works, such as the Keras Dashboard (Chollet, 2015) and TensorBoard (Abadi et al., 2016), have provided real-time visualization features for deep learning model training, enabling users to intuitively observe training progress and performance metrics[5][6]. In this paper, we developed a simple and intuitive web interface for our distributed training system, allowing users to monitor the training process in real time and further reducing the entry barrier.

III. Data

A. Data Source

We used Python's yfinance package to retrieve data[7]. This package fetches publicly available data from Yahoo Finance, allowing flexible combinations of various time resolutions, start times, and end times[8]. Using this package, we obtained historical trading volumes for 39 well-known publicly listed companies with a monthly resolution, setting the start and end dates to the maximum range available from yfinance. After simple filtering of invalid values, we successfully retrieved data covering a period of 50 months, from October 2020 to November 2024. Specific company stock codes, company names, and core business details are provided in table I and table II.

B. Data Format

We defined the following model for the input data objects:

- **Entity** represents the object that generates the data, distinguished by CLASS_ID.
- **Time** represents the moment the data is generated, represented as a tuple (YEAR, MONTH).
- **Event** represents the data generated by an entity at a particular time.
- Each entity should have an event at every time.

The advantage of using this data model is that it decouples the core part of our system from the data source, allowing our system to handle any data object in the $\mathrm{Entity} \times \mathrm{Time}$ space, without being strictly limited to stock trading volume data.

For this data model, we designed the following input format:

- Each line represents one event.
- Each event is represented as a tuple (CLASS_ID, CHANNEL_ID, YEAR, MONTH, UNITS).

The CHANNEL_ID is a reserved field for further scalability considerations. In our current implementation, CHANNEL_ID is filled with the magic number 90 to maintain data consistency. A specific example of the input format can be found in table III.

Due to space constraints, the columns labeled as "CLASS" and "CHANNEL" in the example are actually "CLASS_ID" and "CHANNEL_ID". This input format allows users to dynamically and incrementally input any number of entities and times, providing excellent scalability.

TABLE I
THE STOCK CODES WE USE AND THEIR CORRESPONDING COMPANY DESCRIPTIONS. (FIRST HALF)

Ticker	Company Name Core Business
MSFT	Microsoft Corporation Software, Cloud Computing
AAPL	Apple Inc. Consumer Electronics, Software, Hardware
GOOG	Alphabet Inc. Search Engine, Advertising, Technology
AMZN	Amazon.com, Inc. E-commerce, Cloud Computing, Logistics
META	Meta Platforms, Inc. Social Media, Virtual Reality
NVDA	NVIDIA Corporation GPU, Artificial Intelligence, Semiconductors
ORCL	Oracle Corporation Databases, Cloud Computing
ADBE	Adobe Inc. Creative Software, Digital Media
AMD	Advanced Micro Devices Semiconductors, Processors
INTC	Intel Corporation Semiconductors, Microprocessors
IBM	International Business Machines Enterprise Software, Hardware, Cloud Computing
CRM	Salesforce, Inc. Customer Relationship Management
TSM	Taiwan Semiconductor Manufacturing Company Semiconductor Manufacturing
TSLA	Tesla, Inc. Electric Vehicles, Energy
F	Ford Motor Company Automotive Manufacturing, Financial Services
GM	General Motors Company Automotive Manufacturing, Autonomous Vehicles
LCID	Lucid Group, Inc. Electric Vehicles
NIO	NIO Inc. Electric Vehicles
BYDDF	BYD Company Limited Electric Vehicles, Energy Storage
JPM	JPMorgan Chase & Co. Financial Services, Investment Banking
GS	The Goldman Sachs Group, Inc. Investment Banking, Asset Management
MS	Morgan Stanley Investment Banking, Asset Management
MA	Mastercard Incorporated Payment Solutions
PYPL	PayPal Holdings, Inc. Online Payments
KO	The Coca-Cola Company Beverage Manufacturing
PEP	PepsiCo, Inc. Beverages, Snacks
PG	Procter & Gamble Co. Consumer Goods, Household Products
UL	Unilever PLC Consumer Goods, Household Chemicals
MCD	McDonald's Corporation Fast Food, Restaurants
SBUX	Starbucks Corporation Coffee, Retail
NKE	Nike, Inc. Sportswear, Footwear
DIS	The Walt Disney Company Media, Entertainment
BP	BP p.l.c. Oil, Energy

TABLE II
THE STOCK CODES WE USE AND THEIR CORRESPONDING COMPANY DESCRIPTIONS. (SECOND HALF)

Ticker	Company Name Core Business
JNJ	Johnson & Johnson Medical Devices, Pharmaceuticals
PFE	Pfizer Inc. Pharmaceuticals, Vaccines
MRNA	Moderna, Inc. Biotechnology, Vaccines
CAT	Caterpillar Inc. Heavy Machinery, Engineering Equipment
BA	The Boeing Company Aerospace, Defense
GE	General Electric Company Electrical Equipment, Energy

TABLE III
EXAMPLE OF INPUT DATA FORMAT. IN THE ACTUAL FORMAT, "CLASS" AND "CHANNEL" ARE REPRESENTED AS "CLASS_ID" AND "CHANNEL_ID," RESPECTIVELY.

CLASS string	CHANNEL double	YEAR double	MONTH double	UNITS double
AAPL	90	2020	10	2894666500
AAPL	90	2020	11	2123077300
AAPL	90	2020	12	2322189600
...	...	...	...	...

C. Data Reordering (Spark)

The following steps are implemented within the Spark framework. Since these are still part of the data processing stage, we continue to describe them in the current section.

Our system first reads the uploaded CSV file into a Spark DataFrame object. Then, we perform a series of operations on the DataFrame to rearrange the data.

We use the DataFrame.rdd.map statement to extract the CLASS_ID from each event in each row of the DataFrame. After that, we use the RDD.distinct statement to determine how many entities are present in the input data.

Next, we use the DataFrame.filter statement to sequentially filter out all the events for each entity from the overall DataFrame. This way, we obtain the event table corresponding to each entity.

We then index all the entity event tables by the tuple (YEAR, MONTH) and use DataFrame.join to join them horizontally. As a result, we get a rearranged table with dimensions $(|Time|, |Entity|)$.

D. Data Rescaling (Spark)

We export the rearranged data table and plot it, as shown in fig. 1. We can observe that the magnitudes of the data vary significantly, with some values being very large and others being very small. We need to appropriately scale the data. The scaling transformation should adhere to the following characteristics:

- To avoid relative influence between the data, this transformation should be applied element-wise:

$$f : \mathbb{R} \to \mathbb{R} \tag{1}$$

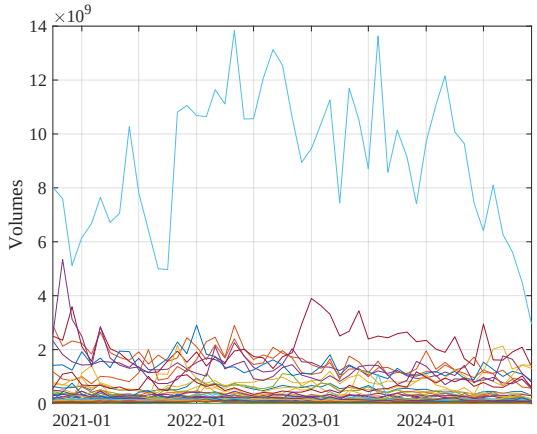

Fig. 1. The original data exhibits a large magnitude difference.

- To ensure that the scaled data preserves the relative order, the transformation should be strictly monotonically increasing:

$$f'(x) > 0, \forall x \in \mathbb{R} \tag{2}$$

- To avoid introducing unnecessary discontinuities that could disrupt the data distribution, the transformation should be smooth:

$$\lim_{\varepsilon \to 0} f'(x+\varepsilon) = f'(x), \forall x \in \mathbb{R} \tag{3}$$

- To maintain symmetry in the value range, the transformation should be symmetric about the zero point:

$$f(x) = -f(-x), \forall x \in \mathbb{R} \tag{4}$$

- To allow for the recovery of the original data from the scaled values, the transformation should be reversible:

$$f^{-1}(f(x)) = x, \forall x \in \mathbb{R} \tag{5}$$

- To unify both large and small magnitudes, the transformation should compress larger input ranges into a smaller output range:

$$\exists M > 0, f(x) - f(-x) < x - (-x) \forall x > M \tag{6}$$

Based on the above discussion, we attempt to construct a transformation that satisfies these properties, while keeping it as simple and direct as possible:

$$f(x) = \begin{cases} \operatorname{sgn}(x) \cdot (1 + \ln|x|), |x| > 1 \\ x, |x| \leq 1 \end{cases} \tag{7}$$

Where sgn refers to the sign function:

$$\operatorname{sgn}(x) = \begin{cases} 1, x > 0 \\ 0, x = 0 \\ -1, x < 0 \end{cases} \tag{8}$$

The inverse transformation of this function can be easily written as:

$$f^{-1}(x) = \begin{cases} \operatorname{sgn}(x) \cdot \exp(|x| - 1), |x| > 1 \\ x, |x| \leq 1 \end{cases} \tag{9}$$

The graph of the mapping $f(x)$ and its inverse $f^{-1}(x)$ can be seen in fig. 2.

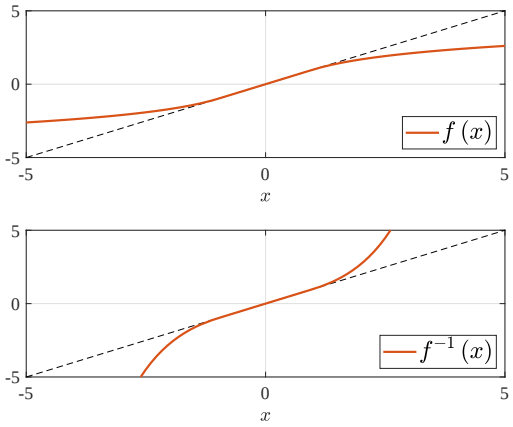

Fig. 2. The mapping $f(x)$ and the inverse mapping $f^{-1}(x)$.

Clearly, this transformation satisfies all the properties discussed above. We apply this transformation element-wise to all the data:

$$x \leftarrow f(x) \tag{10}$$

E. Data Normalization

After the scaling transformation, the magnitudes of our data are now relatively close. We further normalize the data to have a mean of 0 and a standard deviation of 1:

$$x \leftarrow \frac{x - \mu}{\sigma} \tag{11}$$

where μ is the overall mean of the training set (approximately the first 70% of the data), and σ is the overall standard deviation of the training set.

The preprocessed data is plotted and shown in fig. 3.

IV. Methods

A. The Dataset Class

In order to seamlessly integrate our distributed data processing system with the training of PyTorch models, we implemented the torch.Dataset class based on the Spark framework. In this class, we use DataFrame.rdd.zipWithIndex to assign row numbers to the data, and then specifically translate calls from torch.DataLoader into Spark DataFrame calls.

There are two main calls from torch.DataLoader: the __len__ method and the __getitem__ method. For the __len__ method, we translate this call into DataFrame.count(). The translation of the __getitem__ method is more complex. We use DataFrame.filter to filter out the target rows based on the pre-assigned row numbers, and then use DataFrame.collect() to retrieve the actual data content from the DataFrame object.

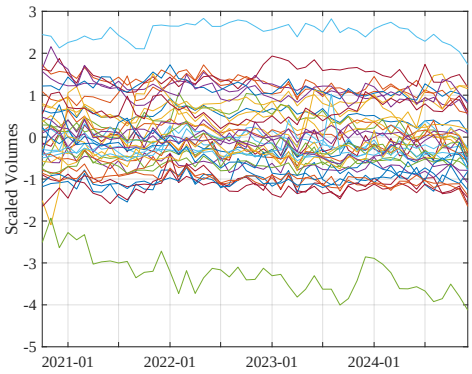

Fig. 3. After rescaling and normalization, the data becomes standardized with a mean of 0, a standard deviation of 1, and consistent magnitudes.

This translation allows us to avoid caching large amounts of data on the training device during torch.DataLoader calls. Instead, the data resides on the distributed cluster and is fetched on-demand in smaller batches when needed.

The working principle of this module is shown in fig. 4.

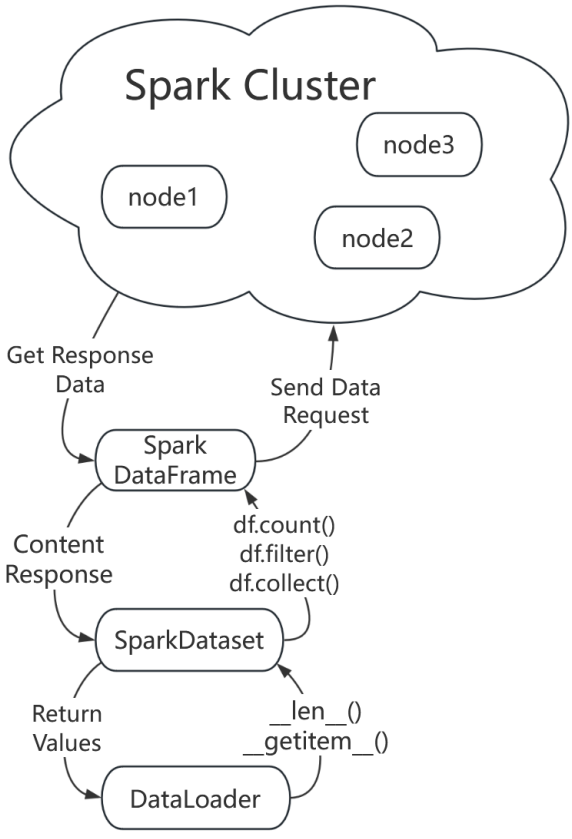

Fig. 4. SparkDataset serves as a translation layer between DataLoader and Spark DataFrame.

B. The Model

Our deep learning model is implemented using GAT (Graph Attention Networks)[9]. In our model, each entity is treated as a graph node, and we read the event data from the four consecutive time steps as the features for each node. Additionally, the model uses the correlation coefficients between these input features as the edge weights of the graph.

After inputting the features, the model outputs a value for each node, representing the predicted values of the nodes at the next time step, immediately following the inputs:

$$\begin{aligned} w_e &= \mathrm{Corr}\left(X_{t-3,\ldots,t}\right) \\ X_{t+1} &= \mathcal{M}\left(X_{t-3,\ldots,t} | w_e\right) \end{aligned} \tag{12}$$

The specific architecture of the model is detailed in fig. 5. We train the model using the setup shown in table IV.

TABLE IV
OPTIONS FOR MODEL TRAINING AND THEIR CORRESPONDING VALUES.

Option	Value
Criterion	MSELoss
Optimizer	SGD
Batch Size	32
Learning Rate	0.005
Momentum	0.9

C. The User Interface

1) The Backend: We use Flask as the web server. The server exposes three main interfaces:

- **/**: When accessing the root path of the service, the server will always return the index.html file of the front-end page.
- **/status**: This path is used to retrieve the current status of the training process, including the model's current prediction results, training loss, reference true values, lists of entity names, and time lists.
- **/draw***: In fact, this involves a series of interfaces, including /draw_plot, /draw_loss, and /draw_heat. These interfaces receive requests with JSON bodies from the front-end and, based on the request content, use matplotlib to generate vector graphics, which are then returned to the front-end.

2) The Frontend: The front-end components consist of multiple polling operations. One polling operation is used to access the /status interface and update the text elements on the page based on the response content. The other three polling operations are used to access /draw_plot, /draw_loss, and /draw_heat. These operations report the front-end user interactions to the back-end and update the images on the page according to the content returned from the back-end. The front-end flowchart is shown in fig. 6.

V. SYSTEM OVERVIEW

Our system consists mainly of a Spark cluster, a deep learning model trainer, a web server, and a front-end. The integration of the entire system is shown in fig. 7. With the

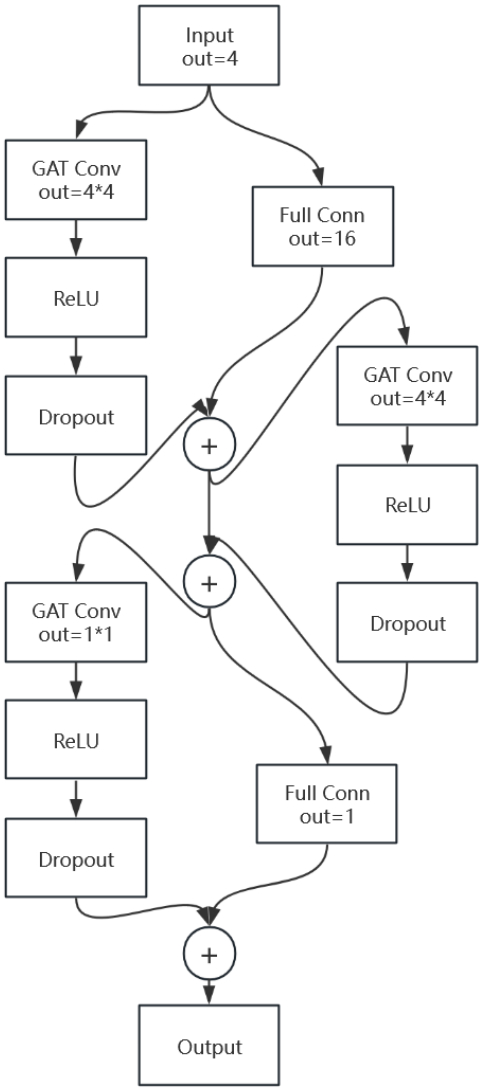

Fig. 5. The GAT model architecture. The entire model is composed of three layers, each containing a GAT convolution and a skip connection. The output sizes of the three layers are 16, 16, and 1, respectively. Between layers of different sizes, the skip connection is implemented using fully connected layers.

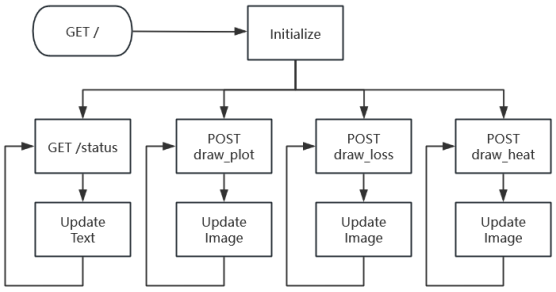

Fig. 6. The Frontend flowchart. After initialization, the frontend will fork four polling processes, each requesting one of the four backend APIs, and update the interface based on the responses.

Docker image we have pre-packaged, users can easily use our system.

Below, we provide detailed instructions for using each part of the system.

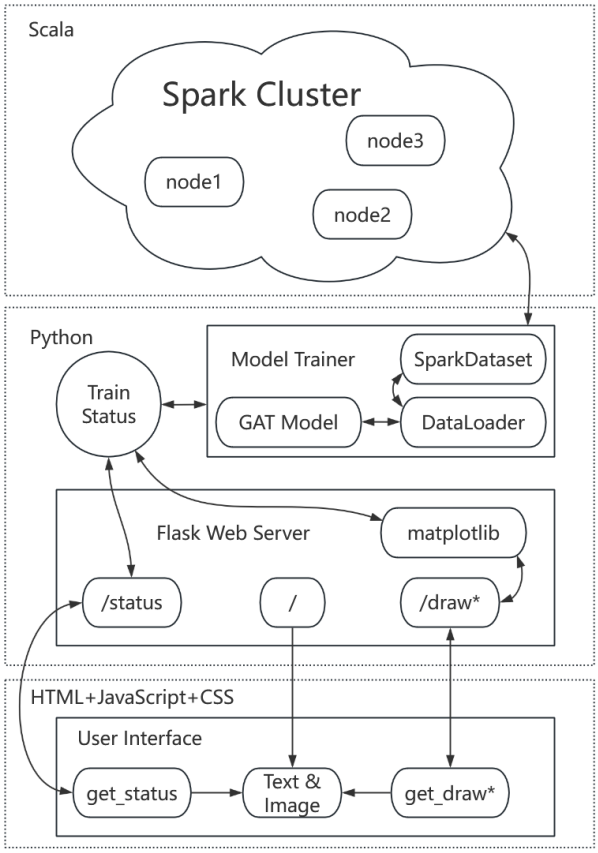

Fig. 7. System integration relationship diagram. The entire system consists of four components: the Spark cluster, model trainer, web server, and user interface.

A. *Spark Cluster*

Please enter the following command in the terminal:

- cd ~/spark−3.5.3−bin−hadoop3
- bash sbin/start−all.sh

You will see an output similar to fig. 8, indicating that the Spark cluster has started successfully.

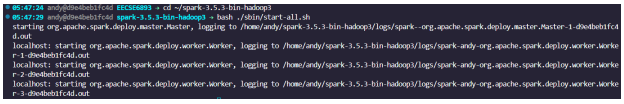

Fig. 8. Start the Spark cluster.

B. *Train and Backend*

Please continue by entering the following command in the terminal:

- cd ~/EECSE6893/
- python3.11 main.py

You will see two parts in the output.

The first part, as shown in fig. 9, indicates that the system is using the Spark framework for data rearrangement, rescaling and normalization.

The second part, as shown in fig. 10, indicates that the system has started training the model.

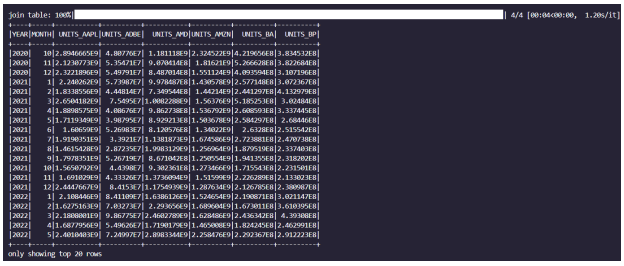

Fig. 9. The system is using the Spark framework for data rearrangement, rescaling and normalization.

Fig. 10. The system is training the model.

C. Frontend

At this point, open the browser and navigate to port 80 to see the frontend page, as shown in fig. 11, fig. 12, fig. 13, and fig. 14:

a) Loss Plot: displays the Mean Squared Error (MSE) loss values on both the training and test sets during each iteration of the training process. See fig. 11.

b) Correlations & Graph: shows the correlations between selected variables (which represent the edge weights in the GAT model) and their approximate node distances. See fig. 12.

c) Predictions: presents the predicted values for the selected variables by the model. See fig. 13.

d) Filter & Table: users can select the stocks they want to view on the filter, and the graph plot, predictions plot, and table will only display the selected items. Each column in the table represents a stock, each row represents a time point, and each cell contains two rows: the first row shows the true value, and the second row shows the predicted value. See fig. 14.

VI. Experiments

A. Real-World Experiment

We first tested using the real dataset described in this paper. We used the Mean Squared Error (MSE) loss as the performance metric for prediction:

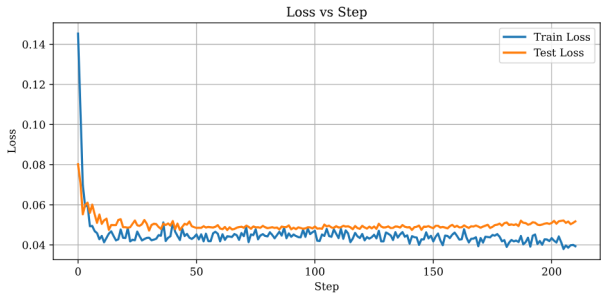

Fig. 11. User Interface: Loss Plot. The horizontal axis represents the training steps, and the vertical axis represents the loss value at each step. The blue line indicates the training set loss, while the orange line indicates the test set loss.

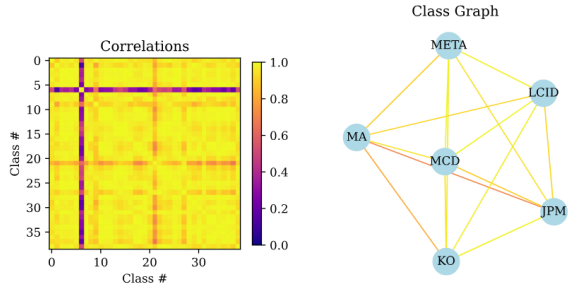

Fig. 12. User Interface: Correlation Heatmap and Class Graph. On the left is the correlation heatmap, where each element represents the edge weight between two graph nodes. On the right is the graph visualization, projected approximately onto a 2D plane based on the edge weights. The nodes displayed on the right are determined by user selection.

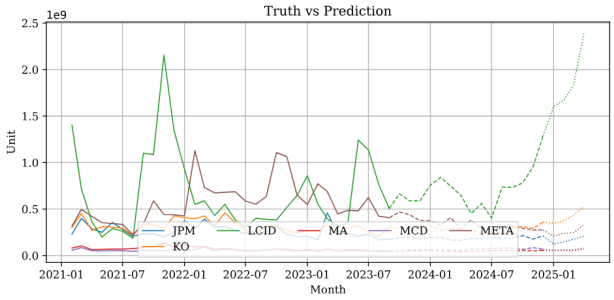

Fig. 13. User Interface: Truth vs Prediction. The horizontal axis represents time, and the vertical axis represents stock trading volume. Each stock is distinguished by a unique color. Solid lines indicate the true values on the training set, while dashed lines represent the predicted values on the test set and extrapolated data. The displayed stocks are determined by user selection.

☐ All
☐ AAPL ☐ ADBE ☐ AMD ☐ AMZN ☐ BA ☐ BP ☐ BYDDF ☐ CAT ☐ CRM ☐ DIS ☐ F ☐ GE ☐ GM
☐ GOOG ☐ GS ☐ IBM ☐ INTC ☐ JNJ ☑ JPM ☑ KO ☑ LCID ☑ MA ☑ MCD ☑ META ☐ MRNA ☐ MS
☐ MSFT ☐ NIO ☐ NKE ☐ NVDA ☐ ORCL ☐ PEP ☐ PFE ☐ PG ☐ PYPL ☐ SBUX ☐ TSLA ☐ TSM ☐ UL

Month	JPM	KO	LCID	MA	MCD	META
2021-02	229,245,300 (269,950,897)	301,062,400 (343,584,184)	1,404,280,700 (231,078,238)	80,667,800 (73,774,626)	57,318,600 (53,302,844)	317,775,200 (360,318,347)
2021-03	399,416,900 (241,750,941)	452,188,000 (305,095,221)	717,979,300 (567,202,487)	103,397,100 (80,984,547)	84,406,200 (56,778,975)	497,416,700 (336,036,828)
2021-04	288,694,200 (298,724,815)	270,807,700 (366,952,128)	364,227,800 (609,084,189)	63,236,700 (83,253,488)	52,201,900 (63,793,507)	421,257,700 (395,971,365)
2021-05	248,957,700 (296,998,655)	309,903,200 (300,597,283)	198,596,900 (529,148,800)	66,245,300 (69,865,914)	49,485,700 (56,503,034)	353,423,700 (408,125,510)

Fig. 14. User Interface: Filter and Table. Users can select which stocks to display using the filter. The filter will not only apply to certain plots but also control which columns are shown in the table.

$$\mathrm{MSE} = \frac{1}{T}\sum_{t} \left\| Y_{t+1} - \mathcal{M}\left(X_{t-3,\ldots,t} | \mathrm{Corr}\left(X_{t-3,\ldots,t}\right)\right)\right\|_2^2 \tag{13}$$

We compared the prediction performance of the GAT model with that of the OLS (Ordinary Least Squares) model and the Constant model. The final test results are shown in table V.

According to the results, our model achieved better prediction performance on both the training and test sets, which demonstrates that the introduction of SparkDataset did not interfere with the training process. This allows us to handle extremely large datasets without negatively affecting the model's training performance.

TABLE V
COMPARISON OF MODEL PREDICTION PERFORMANCE.

Data Set	GAT	OLS	Constant
Train Set	1.1600*	1.4238	2.3814
Test Set	1.2897*	1.6857	3.0466

B. Simulated Load Experiment

We simulated a total of 16GB of synthetic data and conducted simulation training on a PC with 32GB of memory, using the environment shown in table VI. During the simulation training, we successfully completed the model training, and the training speed did not degrade compared to single-machine training on a 32GB memory device. This demonstrates that for commercially large-scale data, our system can effectively distribute the load across multiple devices while maintaining high-speed model training.

TABLE VI
VIRTUAL CLUSTER DEVICE INVENTORY.

Role	vCPU cores	vMem size	Number
Master	2	4GB	1
Worker	2	4GB	6

VII. CONCLUSION

Through the development of this distributed training system, we successfully stored training data in a Spark cluster and achieved seamless integration with Torch's DataLoader, enabling deep learning models to efficiently handle large-scale datasets that exceed the memory capacity of a single machine. Our system leverages Spark's distributed data processing capabilities, using the SparkDataset module to load data on demand from the cluster, effectively reducing memory pressure while ensuring the accuracy of the training process. By combining with a Graph Attention Network (GAT) model, we not only enhanced the ability to process large-scale data but also leveraged the relationships between nodes to improve prediction performance.

Additionally, we developed an intuitive UI for the system, allowing users to monitor the training process in real-time, including loss values, predictions, and model performance. This greatly reduced the learning curve for users and improved the system's usability. Experimental results show that the system outperforms traditional single-machine training in terms of training accuracy and speed, successfully handling real-world datasets as well as simulated massive data.

In the future, we plan to extend the model training algorithms to execute in parallel on distributed clusters. This will not only further improve data throughput but also fully utilize the multi-device computing power of the cluster, achieving training speeds that surpass single-machine capabilities. Moreover, we will continue to optimize the system architecture to enhance its stability and flexibility in large-scale heterogeneous computing environments, meeting the needs of more application scenarios.

REFERENCES

[1] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, pp. 56–65, Oct. 2016.

[2] A. Gupta, H. Thakur, R. Shrivastava, P. Kumar, and S. Nag, "A Big Data Analysis Framework Using Apache Spark and Deep Learning," Nov. 2017. arXiv:1711.09279 [cs].

[3] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang, and G. Song, "BigDL: A Distributed Deep Learning Framework for Big Data," in *Proceedings of the ACM Symposium on Cloud Computing*, (Santa Cruz CA USA), pp. 50–60, ACM, Nov. 2019.

[4] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," Sept. 2018. arXiv:1706.02216 [cs].

[5] F. Chollet *et al.*, "keras," 2015.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," May 2016. arXiv:1605.08695 [cs].

[7] R. Aroussi, "ranaroussi/yfinance," Dec. 2024. original-date: 2017-05-21T10:16:15Z.

[8] "Yahoo Finance - Stock Market Live, Quotes, Business & Finance News."

[9] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," Feb. 2018. arXiv:1710.10903 [stat].