

**DẠI HỌC QUỐC GIA TP. HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**



CHUYÊN ĐỀ 3 TIẾN SĨ

NCS: Phan Hồng Trung

Khóa: 12 – Đợt 2 – 2018

Mã số: N180201

Chuyên Ngành: Công Nghệ Thông Tin (62-48-02-01)

Khoa: Khoa Học & Kỹ Thuật Thông Tin

Tên Chuyên Đề:

**Xây Dựng Nền Tảng Huấn Luyện
Phân Tán Mạng Nơ-ron Sâu**
(Building a Deep Neural Network Distributed
Training Framework)



Cán bộ hướng dẫn: PGS.TS ĐỖ PHÚC

TP. Hồ Chí Minh, tháng 07/2022

MỤC LỤC

DANH MỤC BẢNG	iii
DANH MỤC HÌNH	iv
DANH MỤC THUẬT TOÁN	v
DANH MỤC THUẬT NGỮ	vi
DANH MỤC TỪ VIẾT TẮT	vii
1 TỔNG QUAN	1
1.1 Tổng quan	1
1.2 Các bài toán chính của chuyên đề	2
1.3 Thách thức của bài toán	2
1.4 Những công trình liên quan	2
1.5 Cấu trúc của chuyên đề	3
2 CƠ SỞ LÝ THUYẾT	5
2.1 Cụm máy tính	5
2.2 Spark Cluster	6
2.3 Học sâu	6
2.4 Mạng nơ-ron sâu	7
2.5 Huấn luyện DNN	7
2.5.1 Huấn luyện DNN trong môi trường cục bộ	8
2.5.2 Huấn luyện mô hình DL trong môi trường phân tán	8
3 PHƯƠNG PHÁP LUẬN	14
3.1 Hiện thực hệ thống huấn luyện DNN phân tán trên Apache Spark	14
3.2 Xây dựng DDLF	16
3.2.1 Kiến trúc của DDLF	16

3.2.2	Lớp Request	17
3.2.3	Interface IWorker	18
3.2.4	Lớp Worker	18
3.2.5	Lớp ProxyWorker	19
3.2.6	Lớp Cluster	19
3.3	Triển khai ứng dụng phân tán trên DDLF	20
3.3.1	Triển khai ứng dụng đơn giản	20
3.3.2	Triển khai ứng dụng huấn luyện DNN	21
3.4	Đặc điểm của DDLF	25
3.5	Những hạn chế của Apache Spark và giải pháp trên DDLF	26
3.5.1	Vấn đề tổ chức động dữ liệu và chức năng ở worker nodes	26
3.5.2	Việc xử lý dữ liệu huấn luyện	27
3.5.3	Vấn đề liên lạc bất đồng bộ giữa master node và các worker nodes .	30
3.6	Thách thức trong huấn luyện DNN phân tán	30
4	THỰC NGHIỆM & THẢO LUẬN	33
4.1	Datasets	33
4.2	Cấu hình hệ thống thực nghiệm	33
4.3	So sánh hiệu quả giữa Apache Spark với DDLF	34
5	KẾT LUẬN & HƯỚNG PHÁT TRIỂN	37
	TÀI LIỆU THAM KHẢO	38
	KẾT QUẢ ĐẠT ĐƯỢC	40

DANH MỤC BẢNG

3.1	Các thuộc tính của lớp Request	17
3.2	Các phương thức của interface IWorker	18
3.3	Các phương thức bổ sung của lớp Worker	19
3.4	Các phương thức bổ sung của lớp ProxyWorker	19
3.5	Các phương thức quan trọng của lớp Cluster	19
3.6	Các phương được đặc tả trong interface IApp	22
3.7	So sánh các phương pháp nén dữ liệu	31
4.1	Cấu hình cụm máy tính	33
4.2	Các phần mềm được cài đặt	34
4.3	So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên MNIST	34
4.4	So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên CIFAR10	35
4.5	Tổng kết việc so sánh thời gian huấn luyện trên hai datasets	36

DANH MỤC HÌNH

2.1	Một cụm máy tính	5
2.2	Một mạng nơ-ron sâu	7
2.3	Hai phương pháp huấn luyện phân tán DNN	9
2.4	Huấn luyện phân tán DNN đồng bộ và bất đồng bộ	9
3.1	Kiến trúc của DDL framework	17
3.2	Tải các datasets vào DDLF	30
3.3	So sánh hiệu suất giữa các phương pháp nén dữ liệu	32
3.4	So sánh thời gian xử lý khi áp dụng các phương pháp nén dữ liệu	32
4.1	So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên MNIST	35
4.2	So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên CIFAR10	36

DANH MỤC THUẬT TOÁN

1	Huấn luyện mô hình DL trong môi trường cục bộ	8
2	Huấn luyện phân tán DNN kiểu đồng bộ trên K worker nodes	11
3	Huấn luyện phân tán DNN kiểu bất đồng bộ trên K worker nodes	13
4	Triển khai ứng dụng phân tán đơn giản trên DDLF	20
5	Triển khai ứng dụng huấn luyện mô hình DL trên DDLF	22
6	Huấn luyện phân tán DNN kiểu đồng bộ trên K worker nodes được cải tiến	29

DANH MỤC THUẬT NGỮ

CNN	Convolutional neural network - CNN là mạng nơ-ron tích chập (còn được gọi là ConvNet), là một loại mạng nơ-ron sâu thường được sử dụng để xác định, phân loại, phân tích hình ảnh và video...
HDFS	Hadoop Distributed File System - HDFS là hệ thống tập tin phân tán của Hadoop.
học máy	Machine learning - Học máy là một lĩnh vực của trí tuệ nhân tạo liên quan đến việc nghiên cứu và xây dựng các kỹ thuật cho phép các hệ thống "học" tự động từ dữ liệu để giải quyết những vấn đề cụ thể.
học sâu	Deep learning - Học sâu là tập hợp con của học máy. Học sâu sử dụng các mạng nơ-ron nhiều lớp để đạt được độ chính xác đỉnh cao trong những xử lý phức tạp.

DANH MỤC TỪ VIẾT TẮT

AI	trí tuệ nhân tạo - artificial intelligence.
DDLF	Distributed Deep Learning Framework.
DL	học sâu - deep learning.
DNN	mạng nơ-ron sâu - deep neural network.
ML	học máy - machine learning.
QAS	hệ thống trả lời câu hỏi - query answering system.

Chương 1

TỔNG QUAN

1.1 Tổng quan

Việc huấn luyện *mạng nơ-ron sâu - deep neural network (DNN)* tốn rất nhiều thời gian. Tập dữ liệu huấn luyện càng lớn thì mô hình càng chất lượng nhưng sẽ càng kéo dài thời gian huấn luyện. Thông thường, quá trình huấn luyện kéo dài nhiều ngày hoặc thậm chí vài tuần cho đến khi hội tụ. Điều này thúc đẩy việc phát triển các nền tảng huấn luyện phân tán để tận dụng khả năng tính toán của cụm máy tính. *Apache Spark* là một nền tảng xử lý dữ liệu lớn nổi bật và phổ biến nhất hiện nay. Vì vậy, có nhiều nền tảng huấn luyện *DNN* đã kết hợp với *Apache Spark* để tận dụng đặc điểm tính toán phân tán của nó. Tuy nhiên, trong quá trình phát triển các ứng dụng huấn luyện *DNN* dựa trên *Apache Spark* chúng tôi gặp một số trở ngại. Cấu trúc *map/reduce* của *Apache Spark* rất hữu ích khi xử lý dữ liệu lớn nhưng tỏ ra không phù hợp trong việc huấn luyện *DNN*. Việc kết hợp gượng ép *Apache Spark* để huấn luyện *DNN* làm giảm hiệu suất của toàn hệ thống. Vì vậy, chúng tôi đã phát triển nền tảng học sâu phân tán *Distributed Deep Learning Framework (DDLF)*, hoàn toàn độc lập với *Apache Spark*, khắc phục được những trở ngại mà chúng tôi gặp phải khi dùng *Apache Spark* để huấn luyện mạng nơ-ron.

Các đóng góp trong chuyên đề này có thể được tổng kết như sau:

- Trình bày chi tiết quá trình phát triển nền tảng mới có tên *DDLF* để huấn luyện phân tán *DNN*.
- Mô tả các tính năng của *DDLF* và cách triển khai ứng dụng huấn luyện phân tán *DNN* trên *DDLF*.
- Phân tích những trở ngại khi triển khai hệ thống huấn luyện phân tán *DNN* trên *Apache Spark* và trình bày các giải pháp khắc phục chúng trong *DDLF*.
- So sánh hiệu suất của *DDLF* và *Apache Spark* trong việc huấn luyện phân tán *DNN*.

1.2 Các bài toán chính của chuyên đề

Trong chuyên đề này, chúng tôi đề xuất một giải pháp cho bài toán: “*Xây dựng nền tảng huấn luyện phân tán mạng nơ-ron sâu*”. Bài toán này được chia thành 4 phần:

- Triển khai hệ thống huấn luyện phân tán *DNN* trên *Apache Spark*.
- Phân tích những trở ngại khi triển khai hệ thống huấn luyện phân tán *DNN* trên *Apache Spark* và trình bày các giải pháp khắc phục chúng.
- Phát triển nền tảng mới tên *DDLF* để huấn luyện phân tán *DNN*, hiện thực những giải pháp trên.
- So sánh hiệu suất của *DDLF* và *Apache Spark* trong huấn luyện phân tán *DNN*.

1.3 Thách thức của bài toán

Do nội dung chính của bài toán là xây dựng nền tảng huấn luyện phân tán *DNN*, nên thách thức đầu tiên là phải tìm hiểu về *DNN*, các kỹ thuật huấn luyện *DNN*, kỹ thuật triển khai một hệ thống phân tán.

Tập dữ liệu huấn luyện càng lớn thì mô hình được huấn luyện càng chất lượng nhưng sẽ dẫn đến thách thức phải nghiên cứu kỹ thuật lưu trữ và xử lý dữ liệu lớn. Hơn nữa, để xử lý dữ liệu lớn phân tán trên nhiều máy hiệu quả đòi hỏi phải tối ưu thuật toán và môi trường triển khai.

Mặt khác, để triển khai một hệ thống phân tán đòi hỏi phải đầu tư một hệ thống mạng máy tính với chi phí cao.

1.4 Những công trình liên quan

Nhiều hoạt động nghiên cứu và phát triển các nền tảng huấn luyện phân tán *DNN* đã được thực hiện nhờ sự tiến bộ vượt bậc của học sâu *DL*. Li và cộng sự đã đề xuất một nền tảng máy chủ có tham số cho việc học phân tán và một số phương pháp đã được đề xuất để giảm chi phí liên lạc giữa các nút, chẳng hạn như chỉ trao đổi các giá trị tham số khác 0, lưu trữ cục bộ danh sách chỉ mục và bỏ qua ngẫu nhiên các thông báo đã truyền [1, 2]. Abadi và cộng sự đã trình bày *TensorFlow*, một nền tảng tập trung cho huấn luyện *DNN* tích hợp cả hai phương pháp song song dữ liệu và mô hình [3]. Cả hai công trình đều hỗ trợ liên lạc bất đồng bộ để cải thiện hiệu quả nhưng không kiểm soát tính ổn định của các bản cập nhật *gradient*.

Harlap và cộng sự đã đề xuất một hệ thống pipelined phân tán để huấn luyện *DNN* [4]. Công trình này tập trung vào pipelining với một mô hình song song, phân hoạch các lớp *DNN* trên các máy tính khác nhau và phân chia việc thực thi của các máy tính bằng cách đưa các lô nhỏ (*mini-batches*) liên tiếp vào máy tính đầu tiên. Phương pháp này giảm tải liên lạc vì chỉ các hoạt động và *gradient* của một tập con các lớp được liên lạc giữa các máy tính. Tuy nhiên, cần có các cơ chế phức tạp (như lập hồ sơ, các thuật toán phân hoạch và các giai đoạn được sao lặp) để cân bằng khối lượng công việc giữa các máy tính khác nhau, nếu không, tài nguyên tính toán sẽ bị lãng phí.

Thế hệ tiếp theo của các ứng dụng học máy *ML* sẽ liên tục tương tác với môi trường và học từ những tương tác này. Các ứng dụng này đặt ra các yêu cầu hệ thống mới và khắt khe, cả về hiệu suất và tính linh hoạt. Moritz và cộng sự đã đề xuất *Ray* - một hệ thống phân tán để giải quyết chúng [5]. *Ray* triển khai một giao diện thống nhất có thể đại diện cho cả tính toán song song và tính toán dựa trên tác nhân, được hỗ trợ bởi một công cụ thực thi động duy nhất. Để đáp ứng các yêu cầu về hiệu suất, *Ray* sử dụng bộ lập lịch phân tán và hệ thống lưu trữ chịu lỗi phân tán để quản lý trạng thái điều khiển của hệ thống. Qua thử nghiệm, họ đã chứng minh khả năng mở rộng hơn 1,8 triệu tác vụ mỗi giây và hiệu suất tốt hơn so với các hệ thống chuyên dụng hiện có cho một số ứng dụng học tăng cường đầy thử thách.

Với sự phát triển về kích thước dữ liệu và kích thước mô hình, phương pháp song song dữ liệu không thể hoạt động trên các mô hình có kích thước tham số không thể vừa với bộ nhớ thiết bị có một GPU. Để cải thiện hơn nữa việc huấn luyện mô hình không lồ cấp công nghiệp, Wang và cộng sự đã giới thiệu *Whale*, một nền tảng huấn luyện phân tán thống nhất [6]. Nó cung cấp các chiến lược song song toàn diện bao gồm song song dữ liệu, song song mô hình, đường ống, chiến lược kết hợp và chiến lược song song tự động. *Whale* tương thích với *TensorFlow* và các tác vụ huấn luyện có thể dễ dàng được phân tán bằng cách thêm một vài dòng mã mà không cần thay đổi mã mô hình người dùng. Theo các tác giả, *Whale* là công trình đầu tiên có thể hỗ trợ các chiến lược phân tán kết hợp khác nhau trong một nền tảng. Trong thử nghiệm của họ trên mô hình lớn BERT, chiến lược đường ống của *Whale* nhanh hơn 2,32 lần so với song song dữ liệu của *Horovod* trên 64 GPU. Trong các tác vụ phân loại hình ảnh quy mô lớn (100.000 lớp), chiến lược kết hợp của *Whale*, bao gồm chia nhỏ toán tử và song song dữ liệu, nhanh hơn 14,8 lần so với song song dữ liệu của *Horovod* trên 64 GPU.

Các công trình trên cung cấp nhiều giải pháp đầy triển vọng. Tuy nhiên, chúng không đề cập đến các nội dung sau:

- Phân tích những trở ngại khi triển khai hệ thống huấn luyện phân tán *DNN* trên *Apache Spark* và đề xuất các giải pháp khắc phục chúng.
- Trình bày chi tiết quá trình phát triển nền tảng huấn luyện phân tán *DNN*.

Chính vì vậy, trong chuyên đề này, chúng tôi đề xuất một nền tảng huấn luyện phân tán *DNN*, như là một sự bổ sung vào các nền tảng đang có.

1.5 Cấu trúc của chuyên đề

Cấu trúc của chuyên đề được chia thành 5 chương. Nội dung các chương bao gồm:

1. **CHƯƠNG 1: TỔNG QUAN** trình bày bức tranh khái quát về hiện trạng, khó khăn và tầm quan trọng của vấn đề; định nghĩa bài toán, nêu các thách thức phải giải quyết và khái quát về giải pháp.
2. **CHƯƠNG 2: CƠ SỞ LÝ THUYẾT** trình bày các khái niệm và kỹ thuật quan trọng được sử dụng trong chuyên đề.

3. **CHƯƠNG 3: PHƯƠNG PHÁP LUẬN** trình bày chi tiết về giải pháp xây dựng *QAS*.
4. **CHƯƠNG 4: THỰC NGHIỆM & THẢO LUẬN** trình bày nội dung thực nghiệm và thảo luận về các kết quả thực nghiệm.
5. **CHƯƠNG 5: KẾT LUẬN & HƯỚNG PHÁT TRIỂN** trình bày kết luận chung và hướng phát triển trong tương lai.

Ngoài ra, phần phụ lục trình bày các kết quả đạt được trong quá trình nghiên cứu, và đính kèm là bài báo trình bày một phần của chuyên đề đã được chấp thuận đăng trong tạp chí *Intelligent Data Analysis - IOS Press*.

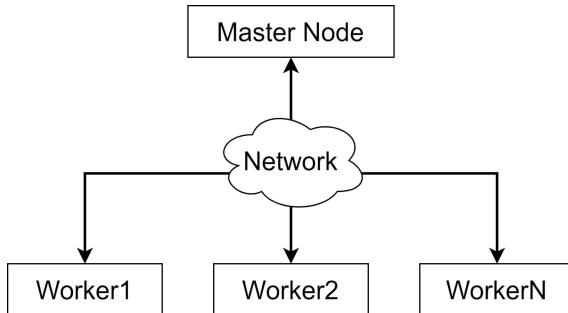
Chương 2

CƠ SỞ LÝ THUYẾT

2.1 Cụm máy tính

Cụm máy tính là nhiều máy tính được kết nối mạng và chúng hoạt động giống như một thực thể duy nhất [7, 8]. Có nhiều kiến trúc cụm máy tính, Hình 2.1 là ví dụ về một cụm tính toán tiêu biểu. Mỗi máy tính tham gia vào cụm được gọi là một node. Trong đó:

- Master node: là máy tính chạy chương trình chính, gửi yêu cầu đến các worker node để thực thi song song và thu thập kết quả.
- Worker node: là máy tính tham gia xử lý các yêu cầu của master node.



Hình 2.1. Một cụm máy tính

Cụm máy tính cung cấp giải pháp xử lý phân tán để giải quyết các vấn đề phức tạp bằng cách chia vấn đề lớn thành nhiều phần nhỏ, mỗi phần nhỏ sẽ được xử lý bởi một worker node. Cụm máy tính có rất nhiều ưu điểm. Sau đây là những ưu điểm tiêu biểu:

- Hiệu quả về chi phí: Cùng một mục đích, thay vì triển khai một máy tính lớn đắt tiền, triển khai một cụm máy tính sẽ kinh tế hơn nhiều.
- Tốc độ xử lý: Các cụm máy tính cung cấp tốc độ xử lý tương tự như tốc độ xử lý của máy tính lớn hay siêu máy tính.

- Tính sẵn sàng cao: Khi một worker node bị lỗi thì cụm máy tính vẫn hoạt động bình thường, dù hiệu suất có giảm đi.
- Khả năng mở rộng: Dễ dàng nâng cấp phần cứng (mở rộng đọc) hoặc thêm/bớt worker node (mở rộng ngang) trong cụm máy tính tùy theo nhu cầu thực tế.
- Tính linh hoạt: Dễ dàng thay đổi số lượng worker node tham gia xử lý một vấn đề cụ thể thông qua việc thay đổi cấu hình.

2.2 Spark Cluster

Có nhiều nền tảng xử lý phân tán như: Hadoop, Apache Spark, Apache Storm, Samza, Flink [9, 10]. Trong đó, Apache Spark là nền tảng xử lý phân tán dữ liệu lớn mạnh mẽ và được dùng phổ biến nhất nhờ các tính năng sau: 1) *Tốc độ xử lý được ví nhanh như chớp*; 2) *Dễ sử dụng và linh hoạt*; 3) *Cung cấp hỗ trợ cho các phân tích phức tạp*; 4) *Xử lý luồng thời gian thực*.

Dể triển khai hệ thống xử lý phân tán trên nền tảng Apache Spark, chúng ta cần phải xây dựng Spark Cluster. Spark Cluster là cụm máy tính mà mỗi máy tính được cài đặt phần mềm Apache Spark để điều khiển hoạt động trong cụm. Do Apache Spark không có hệ thống quản lý file riêng, nên Spark Cluster thường được kết hợp với một hệ thống file phân tán như HDFS của Hadoop (Hadoop Distributed File System - HDFS) hay S3 của Amazon.

Apache Spark là nền tảng tính toán trong bộ nhớ nên các máy trong cụm cần có nhiều bộ nhớ. Hơn nữa, để tạo sự cân bằng và hiệu quả trong xử lý phân tán, cấu hình phần cứng của các worker nodes nên giống nhau. Master node là nơi thu gom tất cả kết quả xử lý từ các worker nodes, thực hiện các xử lý tổng hợp để tạo ra kết quả cuối cùng. Các kết quả gửi về từ worker nodes đôi khi rất lớn, đòi hỏi cấu hình phần cứng của master node phải mạnh hơn các worker nodes.

2.3 Học sâu

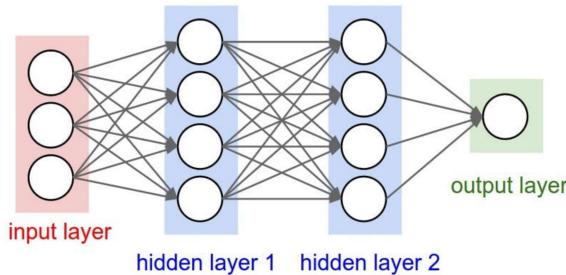
Học sâu - deep learning (DL) là lõi của *học máy - machine learning (ML)* [11] sử dụng nhiều lớp nơ-ron để trích xuất dần dần các đặc trưng ở mức cao hơn từ đầu vào thô. Ví dụ, trong xử lý hình ảnh, các lớp nơ-ron thấp hơn có thể xác định các cạnh, trong khi các lớp nơ-ron cao hơn có thể xác định các khái niệm liên quan đến con người như chữ số, chữ cái hoặc khuôn mặt. Từ "sâu" trong "học sâu" dùng để chỉ số lớp nơ-ron mà dữ liệu được truyền qua đó.

DL khác với *ML* truyền thống ở chỗ nó có thể tự động phát hiện các đặc trưng từ dữ liệu mà không cần sự can thiệp của con người. Nhờ kiến trúc linh hoạt và tiên tiến, *DL* tạo được những đột phá trong lĩnh vực *trí tuệ nhân tạo - artificial intelligence (AI)*. Chương trình cờ vây AlphaGo đánh bại kỳ thủ số 1 thế giới năm 2017, xe tự lái, trợ lý ảo thông minh... là những minh chứng cho những đột phá này. *DL* đang trở thành một trong những lĩnh vực nóng nhất trong khoa học máy tính. Chỉ trong vài năm, *DL* đã thúc đẩy tiến bộ trong nhiều lĩnh vực như phát hiện đối tượng, dịch máy, nhận dạng giọng nói... những vấn đề từng rất khó khăn với các nhà nghiên cứu trí tuệ nhân tạo [12, 13].

2.4 Mạng nơ-ron sâu

Mạng nơ-ron sâu - deep neural network (DNN), như trong Hình 2.2, là một mạng nơ-ron nhân tạo có nhiều lớp nơ-ron ẩn giữa các lớp đầu vào và đầu ra dùng để mô hình hóa các mối quan hệ phi tuyến tính phức tạp.

Mục đích chính của *DNN* là nhận một tập hợp các đầu vào, thực hiện các phép tính phức tạp dần và tạo đầu ra để giải quyết các vấn đề thực tế như phân loại hình ảnh, phân loại văn bản, nhận dạng đối tượng...



Hình 2.2. Một mạng nơ-ron sâu

2.5 Huấn luyện DNN

Việc huấn luyện DNN tốn rất nhiều thời gian do những xử lý phức tạp bên trong. Tập dữ liệu huấn luyện càng lớn thì càng tốn nhiều thời gian huấn luyện. Để giải quyết vấn đề này, chúng ta có hai giải pháp: scale-up (vertical scaling) và scale-out (horizontal scaling) hệ thống [14]:

- Scale-up: một máy tính được bổ sung hoặc nâng cấp tài nguyên (CPU, GPU, TPU, memory, storage, network) để đạt được hiệu suất mong muốn. Scale-up gắn liền với kiến trúc cục bộ nên nó có những ưu điểm và khuyết điểm của kiến trúc cục bộ. Ưu điểm dễ thấy nhất là chi phí trao đổi dữ liệu giữa các luồng xử lý (thread) là không đáng kể. Tuy nhiên, nhược điểm là khả năng nâng cấp tài nguyên bị giới hạn. Ví dụ, việc gắn thêm CPU hoặc bộ nhớ là có giới hạn, không phải gắn thêm bao nhiêu cũng được.
- Scale-out: hệ thống tính toán được bổ sung máy tính để đạt được hiệu suất mong muốn. Scale-out gắn liền với kiến trúc phân tán nên nó có những ưu điểm và khuyết điểm của kiến trúc phân tán. Ưu điểm là việc nâng cấp tài nguyên hầu như không bị hạn chế. Ví dụ, Spark Cluster lớn nhất được biết có tới 8000 nodes [15]. Tuy nhiên, nhược điểm là chi phí liên lạc giữa các worker nodes trong cụm khá lớn, ảnh hưởng đáng kể đến hiệu quả xử lý của toàn bộ hệ thống.

Có hai cách để huấn luyện mô hình DL: huấn luyện cục bộ và huấn luyện phân tán. Nếu tập dữ liệu huấn luyện nhỏ (vừa với bộ nhớ của một máy tính) và mô hình DL đơn giản (ít nơ-ron), chúng ta nên sử dụng huấn luyện cục bộ. Ngược lại, chúng ta nên chọn hình thức huấn luyện phân tán. Tuy nhiên, huấn luyện phân tán phức tạp hơn và phải chịu thêm chi phí liên lạc giữa các máy trong cụm xử lý.

Dộ chính xác của mô hình có thể tăng lên theo số lượng mẫu huấn luyện, số lượng tham số của mô hình, hoặc cả hai. Tuy nhiên, việc huấn luyện các mô hình lớn rất phức tạp và tốn nhiều thời gian khi được huấn luyện trên một máy, ngay cả khi có hỗ trợ đa luồng. Điều này yêu cầu mở rộng việc huấn luyện các mô hình trên nhiều máy tính được kết nối theo cách phân tán.

Để phân tích tầm quan trọng của phương pháp huấn luyện phân tán, trong phần này chúng tôi sẽ trình bày những nội dung sau:

- Huấn luyện DNN trong môi trường cục bộ.
- Huấn luyện DNN trong môi trường phân tán.

2.5.1 Huấn luyện DNN trong môi trường cục bộ

Hiện tại, có rất nhiều nền tảng để triển khai DL. Năm nền tảng tốt nhất có thể kể là TensorFlow, Keras, PyTorch, Apache MXNet và Microsoft Cognitive Toolkit [16]. Trong đó, Keras đơn giản và dễ sử dụng nhất. Đặc điểm nổi bật của Keras bao gồm: 1) API dễ hiểu và nhất quán; 2) Có thể tích hợp với nhiều nền tảng DL bên dưới như: TensorFlow, Theano và CNTK; 3) Hỗ trợ huấn luyện phân tán và song song trên nhiều GPU. Bên cạnh đó, TensorFlow là nền tảng tốt nhất và được ưa thích nhất hiện nay. Đặc điểm nổi bật của Tensorflow gồm: 1) Hỗ trợ nhiều GPU mạnh mẽ; 2) Trực quan hóa đồ thị tính toán; 3) Tài liệu và hỗ trợ cộng đồng tuyệt vời. Tuy nhiên, Tensorflow được xem là nền tảng phức tạp, khó sử dụng. Chính vì vậy, Keras đã được chọn làm API cấp cao của Tensorflow 2 nhằm dễ tiếp cận và đạt hiệu quả cao khi giải quyết các vấn đề DL. Từ những phân tích trên, chúng tôi chọn Keras làm nền tảng huấn luyện mô hình DL trong môi trường cục bộ trong nghiên cứu này.

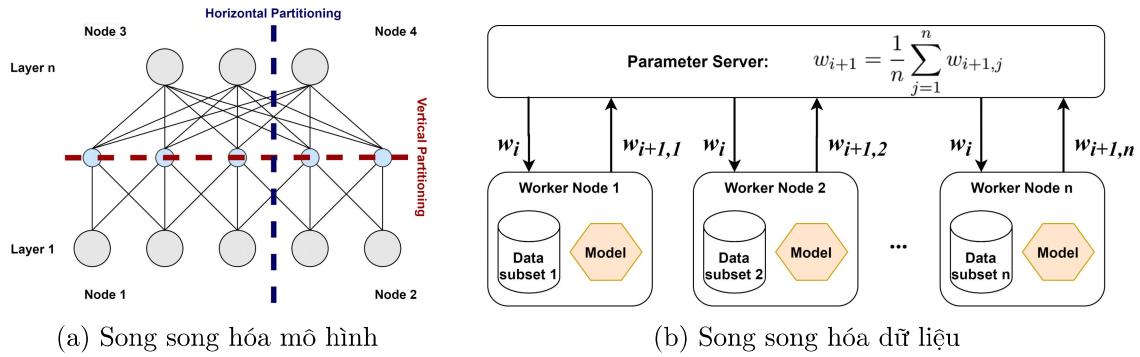
Thuật toán 1 trình bày khái quát quá trình huấn luyện mô hình DL ở chế độ cục bộ bằng Keras. Đầu tiên, hai tập dữ liệu huấn luyện và kiểm thử được tải vào hệ thống. Sau đó, mô hình DL sẽ được tạo theo nhu cầu. Trước khi huấn luyện, cần cấu hình quá trình học bằng phương thức *model.compile()*. Tiếp theo, gọi phương thức *model.fit()* để tiến hành quá trình huấn luyện mô hình bằng tập dữ liệu huấn luyện. Cuối cùng, đánh giá mô hình bằng phương thức *model.evaluate()* với tập dữ liệu kiểm thử. Lưu ý, các tham số trong thuật toán này có thể thay đổi tùy theo nhu cầu thực tế.

Thuật toán 1: Huấn luyện mô hình DL trong môi trường cục bộ

```
1 (x_train, y_train), (x_test, y_test) ← load_dataset()  
2 model ← create_model()  
3 model.compile(loss='categorical_crossentropy', optimizer='sgd',  
    metrics=['accuracy'])  
4 model.fit(x_train, y_train, epochs=10, batch_size=32)  
5 loss_and_metrics = model.evaluate(x_test, y_test, batch_size=32)
```

2.5.2 Huấn luyện mô hình DL trong môi trường phân tán

Có hai cách huấn luyện phân tán chính: song song hóa mô hình (model parallelism) và song song hóa dữ liệu (data parallelism) [17–19].

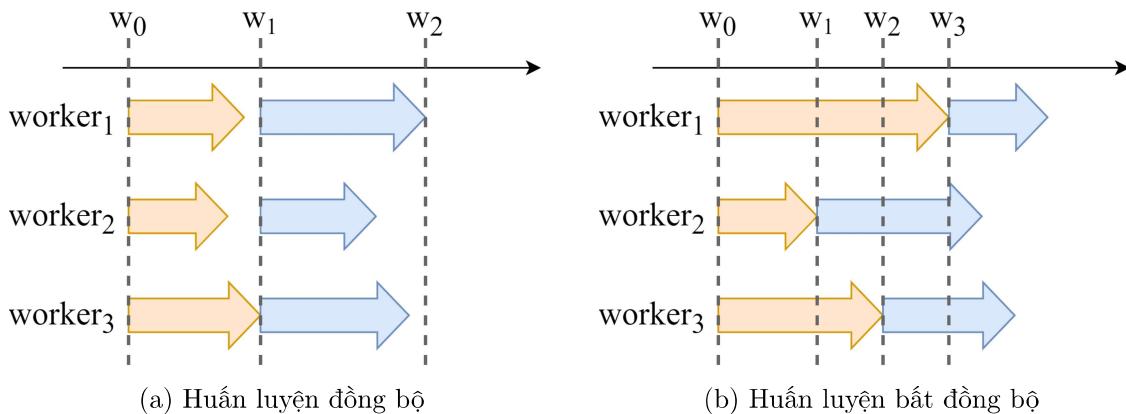


Hình 2.3. Hai phương pháp huấn luyện phân tán DNN

Trong song song hóa mô hình, như trong Hình 2.3a, mô hình được chia thành các phần khác nhau để có thể chạy đồng thời ở các máy khác nhau và mỗi phần sẽ chạy trên cùng một tập dữ liệu. Các worker nodes chỉ cần đồng bộ hóa các trọng số được dùng chung, thường là một lần cho mỗi bước lan truyền tiến hoặc lùi. Do phương pháp này khá phức tạp, chúng ta chỉ nên dùng khi mô hình lớn không vừa trong một máy. Vì tính chất phức tạp của phương pháp này và phạm vi giới hạn của chuyên đề nên trong nghiên cứu này chúng tôi chỉ tập trung vào phương pháp song song hóa dữ liệu.

Song song hóa dữ liệu, như trong Hình 2.3b, là phương pháp phổ biến nhất, dễ thực hiện nhất và phù hợp với hầu hết mọi trường hợp. Trong phương pháp này, dữ liệu được chia thành các phân vùng và được phân phối cho các worker nodes. Mô hình được tạo ở master node, sau đó được gửi đến các worker nodes và được huấn luyện một cách độc lập bằng các phân vùng dữ liệu đã phân phối trước đó. Sau khi huấn luyện mô hình trong một khoảng thời gian có kiểm soát, các trọng số của mô hình được worker nodes gửi về master node và được tính trung bình để cập nhật tập trọng số cho mô hình ở master node.

Phương pháp song song hóa dữ liệu bao gồm hai loại chính: 1) Huấn luyện đồng bộ (Hình 2.4a): tất cả các worker nodes đồng bộ với nhau trong quá trình huấn luyện; 2) Huấn luyện bất đồng bộ (Hình 2.4b): mỗi máy huấn luyện theo tốc độ riêng của nó.



Hình 2.4. Huấn luyện phân tán DNN đồng bộ và bất đồng bộ

A. Huấn luyện DNN kiểu đồng bộ

Trong huấn luyện đồng bộ, hệ thống huấn luyện phải đợi tất cả worker nodes hoàn thành đợt huấn luyện, gradients của chúng được gửi về master node để cập nhật mô hình chính, master node sẽ gửi bộ trọng số mới đến tất cả worker nodes để cập nhật mô hình của chúng cùng một lúc. Phương pháp này có một nhược điểm là không tận dụng được hết khả năng của cụm tính toán do các worker nodes nhanh phải đợi các worker nodes chậm. Để hạn chế tối thiểu nhược điểm này, cấu hình của các worker nodes phải giống nhau.

Thuật toán 2 trình bày quá trình huấn luyện DNN kiểu đồng bộ. Hàm *master_train()* được thực thi trên master node, và hàm *worker_j_train()* được thực thi trên *worker_j*. Trên master node, đầu tiên, tập dữ liệu được tải vào hệ thống. Kế tiếp, master node tạo *model* và gửi đến các worker nodes. Trong vòng lặp huấn luyện với số epochs là M , tập dữ liệu được xáo trộn, được phân hoạch thành K partitions (K là số lượng worker node trong cụm máy tính), và partition P_j được gửi đến *worker_j*. Sau đó, bộ trọng số w được gửi đến tất cả worker nodes để cập nhật và huấn luyện trên các worker nodes. Khi tất cả các workers hoàn thành đợt huấn luyện, master node thu thập các gradients và cập nhật mô hình chính. Nếu mô hình đã thỏa mãn điều kiện hội tụ, master node sẽ gửi tín hiệu *STOP* đến các worker nodes để ngừng quá trình huấn luyện. Ở *worker_j*, nó đợi nhận *model* từ master node để tiến hành quá trình huấn luyện. Trong vòng lặp huấn luyện, *worker_j* đợi nhận partition P_j và bộ trọng số w_j từ master node, cập nhật *model_j*, và bắt đầu quá trình huấn luyện. Sau khi huấn luyện xong N epochs, gradients g_j được gửi về master node để cập nhật mô hình chính. Nếu nhận được tín hiệu *STOP* từ master node, *worker_j* sẽ kết thúc quá trình huấn luyện, ngược lại nó sẽ tiếp tục đợt huấn luyện kế tiếp cho đến khi đủ M epochs.

Thuật toán 2: Huấn luyện phân tán DNN kiểu đồng bộ trên K worker nodes

```

function master_train()
1   dataset ← load_dataset()
2   model ← create_model()
3   broadcast_model(model)
4   w ← model.get_weights()
5   i ← 0
6   while  $i < M$  do
7     dataset ← shuffle(dataset)
8      $\cup_{j=1}^K P_j$  ← partition(dataset)
9     send_data( $\cup_{j=1}^K P_j$ )
10    broadcast_weights(w)
11     $g \leftarrow \text{await collect}(g_j) \text{ from all workers}$ 
12     $w \leftarrow w - \frac{1}{K} \times \sum_{j=1}^K g_j$ 
13    model.set_weights(w)
14    if is_convergent(model) then
15      broadcast_signal(STOP)
16      break
17     $i \leftarrow i + N$ 

```

```

function workerj_train()
1    $model_j \leftarrow \text{await receive\_model}$ ()
2   i ← 0
3   while  $i < M$  do
4      $P_j \leftarrow \text{await receive\_data}$ ()
5      $w_j \leftarrow \text{await receive\_weights}$ ()
6      $model_j.set\_weights(w_j)$ 
7      $model_j.fit(P_j, epochs = N, batch\_size = BS)$ 
8      $g_j \leftarrow model_j.get\_gradients()$ 
9     send_gradients( $g_j$ ) to the master node
10     $i \leftarrow i + N$ 
11    if has_signal(STOP) then
12      break

```

B. Huấn luyện DNN kiểu bất đồng bộ

Trong phương pháp huấn luyện bất đồng bộ, khả năng của cụm máy tính được tận dụng tối đa do không có hiện tượng các worker node đợi nhau. Trong phương pháp này, khi một worker node hoàn thành đợt huấn luyện, gradients của nó được gửi về master node để cập nhật mô hình chính, bộ trọng số mới sẽ được gửi trở lại worker node đó để tiến hành đợt huấn luyện kế tiếp. Ngoài việc dùng gradients cập nhật mô hình chung ở master node, các worker nodes hoạt động hoàn toàn độc lập với nhau. Điều này dẫn đến một hệ quả là các worker nodes khác không được cập nhật kịp thời bộ trọng số mới, tiếp tục sử dụng bộ trọng số cũ đến hết đợt huấn luyện và có thể đi chệch mục tiêu. Thuộc

tính này được gọi là *staleness*. Một cách để giảm bớt hiện tượng này là thay đổi tốc độ học tùy thuộc vào thuộc tính *staleness* - một tốc độ học nhỏ hơn đối với các worker nodes cũ hơn. Bằng cách này, khi worker node di chêch hướng, mức độ chêch hướng tương đối nhỏ.

Thuật toán 3 trình bày quá trình huấn luyện DNN kiểu bất đồng bộ. Tương tự như kiểu huấn luyện đồng bộ, đầu tiên master node cung cấp dữ liệu, tạo mô hình, gửi mô hình đến các worker nodes. Do các worker nodes hoạt động độc lập nhau, master node sẽ tạo một thread để điều khiển việc huấn luyện trên một worker node. Xem xét $thread_j$, đầu tiên, tập dữ liệu được xáo trộn, được phân hoạch, và phân hoạch P_j được gửi đến $worker_j$. Sau đó, master node gửi bộ trọng số đến $worker_j$ để huấn luyện, và đợi nhận gradients kết quả để cập nhật mô hình chính. Do mô hình chính được dùng chung bởi K threads nên trước khi cập nhật, $thread_j$ phải khóa độc quyền mô hình chính để tránh sự xung đột giữa các threads. Hoạt động huấn luyện ở các worker nodes diễn ra giống như trong phương pháp huấn luyện đồng bộ.

Thuật toán 3: Huấn luyện phân tán DNN kiểu bất đồng bộ trên K worker nodes

```

function master_train()
1   dataset ← load_dataset()
2   model ← create_model()
3   broadcast_model(model)
4   w ← model.get_weights()
5   for  $j \leftarrow 1$  to  $K$  do
6      $w_j \leftarrow w$ 
7     run_thread( $j$ )
8      $i \leftarrow 0$ 
9     while  $i < M$  do
10    dataset ← shuffle(dataset)
11     $\cup_{j=1}^K P_j \leftarrow partition(dataset)$ 
12    send_data( $P_j$ ) to worker $_j$ 
13    send_weights( $w_j$ ) to worker $_j$ 
14     $g_j \leftarrow await collect(g_j)$  from worker $_j$ 
15    lock(model)
16     $w_j \leftarrow model.get_weights()$ 
17     $w_j \leftarrow w_j - \frac{1}{K} \times g_j$ 
18    model.set_weights( $w_j$ )
19    if is_convergent(model) then
20      broadcast_signal(STOP)
21      break
22     $i \leftarrow i + N$ 

```

```

function worker $_j$ _train()
1   model $_j \leftarrow await receive\_model()$ 
2    $i \leftarrow 0$ 
3   while  $i < M$  do
4      $P_j \leftarrow await receive\_data()$ 
5      $w_j \leftarrow await receive\_weights()$ 
6     model $_j.set\_weights(w_j)$ 
7     model $_j.fit(P_j, epochs = N, batch\_size = BS)$ 
8      $g_j \leftarrow model_j.get\_gradients()$ 
9     send_gradients( $g_j$ ) to the master node
10     $i \leftarrow i + N$ 
11    if has_signal(STOP) then
12      break

```

Chương 3

PHƯƠNG PHÁP LUẬN

3.1 Hiệu thực hệ thống huấn luyện DNN phân tán trên Apache Spark

Apache Spark là nền tảng xử lý phân tán chuyên dùng xử lý và phân tích dữ liệu lớn rất hiệu quả. Nhờ có nhiều ưu điểm, Apache Spark đã trở thành nền tảng xử lý dữ liệu lớn được dùng phổ biến nhất hiện nay. Tuy nhiên, khi triển khai ứng dụng huấn luyện DNN phân tán, Apache Spark cho thấy có nhiều nhược điểm, gây khó khăn trong việc triển khai ứng dụng và ảnh hưởng xấu đến toàn bộ hệ thống. Ví dụ 3.1 trình bày việc hiện thực Thuật toán 2 để huấn luyện DNN phân tán kiểu đồng bộ trên Apache Spark. Trong hiện thực này, chúng tôi kết hợp Keras với Apache Spark.

Ví dụ 3.1. Hiện thực Thuật toán 2 trên Apache Spark

```
1 def master_train():
2     # Load the dataset
3     (x_train, y_train), (x_test, y_test) = load_dataset()
4     # Create the model
5     model = create_model()
6     # Broadcast the yaml string to workers for restoring the model
7     bc_yaml = sc.broadcast(model.to_yaml())
8     weights = model.get_weights()
9     i = 0
10    while(i < M):
11        # Shuffle the training data
12        permutation = np.random.permutation(len(x_train))
13        x_train = x_train[permutation]
14        y_train = y_train[permutation]
15        # Build RDD from the training data and partition the RDD into K partitions
16        rdd = build_rdd(sc, x_train, y_train, K)
17        # Broadcast weights to workers
18        bc_weights = sc.broadcast(weights)
19        # Call worker_train() on workers and await to collect gradients from workers
20        gradients = rdd.mapPartitions(worker_train).collect()
```

```
21     # Update weights of the model at the master node
22     for gradient in gradients:
23         weighted_grad = divide(gradient, K)
24         weights = subtract(weights, weighted_grad)
25         model.set_weights(weights)
26         i += N
27
28 def worker_train(data_iterator):
29     # Restore the model from a yaml string
30     model = model_from_yaml(bc_yaml.value)
31     # Load x_train, y_train from the data_iterator
32     x_train, y_train = load_partition(data_iterator)
33     # Compile the model
34     compile(model)
35     # Get weights from the master node
36     weights = bc_weights.value
37     # set weights to the model
38     model.set_weights(weights)
39     # Train the model
40     model.fit(x_train, y_train, epochs = N, batch_size = BS)
41     # calculate gradients
42     new_weights = model.get_weights()
43     gradients = subtract(weights, new_weights)
44     yield gradients
```

Trong hiện thực trên có ba vấn đề lớn, ảnh hưởng xấu đến hiệu suất của toàn bộ hệ thống: 1) Trong mỗi lần lặp, việc xáo trộn dữ liệu, phân hoạch lại dữ liệu và gửi các phân hoạch mới đến các worker nodes là xử lý tốn nhiều thời gian; 2) Tự duy lập trình bị phụ thuộc vào cấu trúc map/reduce, người lập trình không được tự do phát triển ý tưởng; 3) Hàm `worker_train()` được gửi từ master node đến các worker nodes nhiều lần, trong khi nội dung của hàm này không thay đổi, làm tăng lưu lượng trên mạng không cần thiết. Dựa trên góc nhìn khi phát triển ứng dụng huấn luyện DNN phân tán, chúng tôi nhận thấy Apache Spark có các nhược điểm như sau:

- Khi huấn luyện DNN phân tán, việc xáo trộn dữ liệu, phân hoạch lại dữ liệu và gửi các phân hoạch mới đến các worker nodes là không thể tránh khỏi. Nhưng đây lại là những thao tác tốn nhiều thời gian xử lý trên Apache Spark. Điều này làm ảnh hưởng đến hiệu quả của toàn bộ hệ thống huấn luyện DNN phân tán.
- Cấu trúc `map/reduce` của Apache Spark được áp dụng rất hiệu quả trong xử lý dữ liệu lớn nhưng không phù hợp trong việc huấn luyện DNN phân tán vì việc lập trình bị gò bó trong cấu trúc `map/reduce`, trói buộc tự duy lập trình, dẫn đến việc triển khai ứng dụng huấn luyện DNN kém linh hoạt và không hiệu quả.
- Do phụ thuộc vào cấu trúc `map/reduce`, hàm `worker_train(data_iterator)` chỉ nhận một tham số duy nhất là `data_iterator` (là 1 data partition). Người lập trình không thể truyền thông tin khác cho hàm `worker_train()` khi cần. Dùng biện pháp thay thế thì quá phiền phức. Điều này làm khó triển khai ứng dụng.

- Cơ chế chia sẻ dữ liệu giữa master node và worker nodes rất hạn chế thông qua *broadcast* và *accumulated variables*. Trong khi broadcast variables là biến chỉ đọc, còn accumulated variables là biến chỉ cộng.
- Người sử dụng không thể chủ động thêm/bớt biến hay phương thức vào worker node.
- Spark không hỗ trợ liên lạc bất đồng bộ giữa master node và các worker nodes. Do đó, để triển khai ứng dụng huấn luyện DNN kiểu bất đồng bộ, người phát triển ứng dụng phải tự bổ sung cơ chế liên lạc bất đồng bộ vào Apache Spark hoặc sử dụng một framework có hỗ trợ.

3.2 Xây dựng DDLF

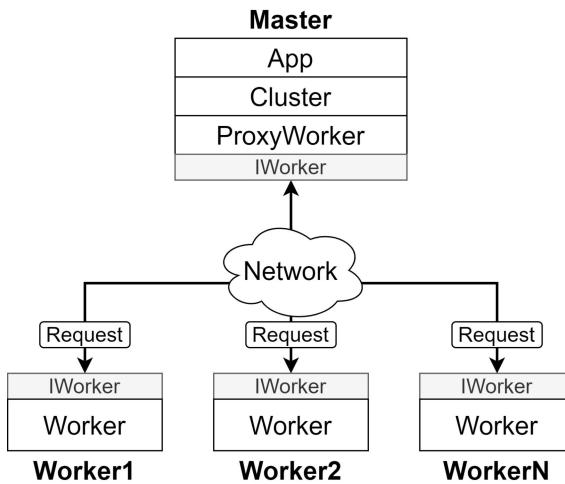
Dể tránh những nhược điểm trên của Apache Spark khi phát triển ứng dụng huấn luyện DNN phân tán, chúng tôi cân nhắc 2 giải pháp: 1) Sử dụng framework có sẵn; 2) Xây dựng framework mới. Hiện tại, có nhiều framework hỗ trợ huấn luyện DNN phân tán như: Distributed TensorFlow[3], Elephas, Horovod, TensorFlowOnSpark, BigDL, PyTorch, Ray. Dùng framework có sẵn giúp phát triển nhanh ứng dụng nhưng mặt trái là phải chấp nhận những khuyết điểm nội tại của nó và khó cải tiến, dẫn đến kém linh hoạt. Chúng tôi chọn giải pháp xây dựng framework mới vì các lý do sau đây:

- Không phụ thuộc vào framework có sẵn: Thật là khó chịu khi dùng framework bị lỗi mà không biết rõ nguyên nhân. Hơn nữa, biết nguyên nhân lỗi nhưng không thể khắc phục do những khuyết điểm thiết kế và kỹ thuật áp dụng không thay đổi được.
- Làm chủ công nghệ: Framework do chúng tôi phát triển nên chúng tôi hiểu rõ các kỹ thuật được áp dụng, hiểu rõ bản chất hoạt động bên trong của nó, làm chủ những kỹ thuật then chốt.
- Làm sâu sắc nội dung nghiên cứu: Khi phát triển framework, chúng tôi phải tiến hành nhiều nghiên cứu, khảo sát, làm sâu sắc thêm nội dung đang nghiên cứu.
- Linh hoạt: Chúng tôi không bị gò bó vào những cấu trúc lập trình có sẵn, chẳng hạn như *map/reduce*, giúp giải phóng tư duy lập trình, tự do, thoả mái phát triển ý tưởng.
- Khả năng mở rộng: hiểu rõ tường tận kỹ thuật giúp dễ dàng cải tiến và mở rộng framework.
- Khắc phục những hạn chế đã gặp: thông qua kinh nghiệm sử dụng các framework có sẵn, chúng tôi có thể cung cấp giải pháp cho các vấn đề đã gặp.

3.2.1 Kiến trúc của DDLF

Kiến trúc của DDLF là cụm máy tính gồm một master node và nhiều worker nodes như Hình 3.1. Master node sẽ điều khiển các worker nodes để xử lý tác vụ theo kiểu phân tán. Interface IWorker đặc tả chức năng của worker nodes. Các worker nodes được hiện

thực bằng class Worker. Mỗi master node bao gồm 3 thành phần chính: 1) ProxyWorker: đại diện cho worker node; 2) Cluster: biểu diễn cluster; 3) App: biểu diễn một ứng dụng xử lý phân tán. Lớp *Request* biểu diễn một yêu cầu Remote Procedure Call (RPC) được gửi từ lớp master node đến các worker nodes.



Hình 3.1. Kiến trúc của DDL framework

3.2.2 Lớp Request

Lớp Request biểu diễn một yêu cầu RPC. Bảng 3.1 mô tả các thuộc tính của lớp Request.

Bảng 3.1. Các thuộc tính của lớp Request

#	Thuộc Tính	Mô Tả
1.	command	Chuỗi ký tự được dùng để xác định yêu cầu.
2.	args	Tập hợp các tham số loại <i>Non Keyword Arguments</i> cần khi xử lý yêu cầu.
3.	kwargs	Tập hợp các tham số loại <i>Keyword Arguments</i> cần khi xử lý yêu cầu.

Ví dụ, để gửi phương thức *add()* đến các worker node và thực thi, master node phải tạo đối tượng của lớp *Request* như trong Ví dụ 3.2. Trong đó, thuộc tính *command = 'run_method'* xác định yêu cầu là “gửi phương thức đến worker node để thực thi”; thuộc tính *args = ['add']* xác định phương thức được gửi đến worker node là phương thức *add()*; và thuộc tính *kwargs = {a=20, b=60}* cung cấp các tham số cho phương thức *add()* khi thực thi.

Ví dụ 3.2. Ví dụ tạo yêu cầu “gửi phương thức *add()* đến worker node để thực thi”

```
1 def add(a, b):
```

```

2  return a + b
3
4  request = Request('run_method', add, a=20, b=60)

```

3.2.3 Interface IWorker

Interface IWorker đặc tả chức năng cho các worker nodes và được dùng làm giao diện liên lạc giữa master node và các worker nodes. Interface này được hiện thực bởi lớp Worker và lớp ProxyWorker. Bảng 3.2 mô tả các chức năng chính của interface IWorker.

Bảng 3.2. Các phương thức của interface IWorker

#	Phương Thức	Mô Tả
1.	async def connect(self)	Mở kết nối TCP (Transmission Control Protocol) giữa master node và worker node.
2.	async def close(self)	Đóng kết nối.
3.	async def load_cifar10(self)	Tải dataset CIFAR10.
4.	async def load_mnist(self)	Tải dataset MNIST.
5.	async def add_method(self, method_code, method_name)	Thêm một phương thức vào worker node.
6.	async def remove_method(self, method_name)	Xóa một phương thức ở worker node.
7.	async def run(self, method_name, **kwargs)	Thực thi một phương thức ở worker node.
8.	async def run_code(self, code)	Gửi code đến worker node và thực thi đoạn code đó ở worker node.
9.	async def run_method(self, method_code, method_name, **kwargs)	Gửi một phương thức đến worker node và thực thi phương thức đó ở worker node.
10.	async def shutdown(self)	shut down the worker node.

3.2.4 Lớp Worker

Lớp Worker hiện thực interface IWorker và định nghĩa các phương thức được thực thi ở một worker node. Ngoài các phương thức được đặc tả trong interface IWorker, lớp Worker có định nghĩa thêm một số phương thức xử lý đặc thù được mô tả trong Bảng 3.3.

Bảng 3.3. Các phương thức bổ sung của lớp Worker

#	Phương Thức	Mô Tả
1.	def start(self)	Khởi động worker node.
2.	async def control(self, reader: StreamReader, writer: StreamWriter)	Thực hiện vòng lặp: nhận yêu cầu từ master node → xử lý yêu cầu → gửi kết quả về master node.
3.	async def handle(self, req: Request)	Xử lý một yêu cầu của master node.

3.2.5 Lớp ProxyWorker

Một đối tượng của lớp *ProxyWorker* đại diện cho một worker node ở phía master node. Do đó lớp *Worker* và *ProxyWorker* cùng hiện thực một interface chung là *IWorker*. Ngoài những phương thức được đặc tả trong interface *IWorker*, lớp *ProxyWorker* cũng bổ sung những phương thức riêng của nó như trong Bảng 3.4.

Bảng 3.4. Các phương thức bổ sung của lớp ProxyWorker

#	Phương Thức	Mô Tả
1.	async def connect(self)	Tạo kết nối với worker node.
2.	async def rpc(self, req)	Thực hiện gọi phương thức từ xa.

3.2.6 Lớp Cluster

Lớp *Cluster* biểu diễn một cluster gồm nhiều worker nodes. Lớp này cung cấp các chức năng mà cluster đảm nhận. Các phương thức của lớp *Cluster* tương tự như các phương thức được đặc tả trong interface *IWorker* nhưng khác nhau ở chỗ: các phương thức trong interface *IWorker* đặc tả các chức năng của một worker node, còn các phương thức của lớp *Cluster* đặc tả các chức năng của một cluster. Bảng 3.5 mô tả các phương thức quan trọng của lớp *Cluster*.

Bảng 3.5. Các phương thức quan trọng của lớp Cluster

#	Phương Thức	Mô Tả
1.	async def connect(self)	Tạo kết nối giữa master node với các worker nodes trong cluster.
2.	async def close(self)	Dóng kết nối giữa master node với các worker nodes trong cluster.
3.	async def add_method(self, method)	Thêm một phương thức vào các worker nodes trong cluster.

#	Phương Thức	Mô Tả
4.	async def remove_method(self, method)	Xóa một phương thức khỏi các worker nodes trong cluster.
5.	async def run(self, method, **kwargs)	Thực thi một phương thức ở các worker nodes trong cluster.
6.	async def run_at(self, worker, method, **kwargs)	Thực thi một phương thức ở một worker node trong cluster.
7.	async def run_code(self, code)	Gửi một đoạn code đến các worker nodes trong cluster và thực thi đoạn code đó ở các worker nodes.
8.	async def run_method(self, method, **kwargs)	Gửi một phương thức đến các worker nodes trong cluster và thực thi phương thức đó ở các worker nodes.
9.	async def shutdown(self)	shutdown các worker nodes trong cluster, hay nói cách khác là shutdown cluster.

3.3 Triển khai ứng dụng phân tán trên DDLF

3.3.1 Triển khai ứng dụng đơn giản

Để triển khai một ứng dụng phân tán đơn giản, chúng ta thực hiện các bước như trong Thuật toán 4.

Thuật toán 4: Triển khai ứng dụng phân tán đơn giản trên DDLF

- 1 *Đọc cấu hình của cluster*
 - 2 *Tạo đối tượng của lớp Cluster*
 - 3 *Kết nối master node với cluster*
 - 4 *Thực hiện các xử lý phân tán trên cluster*
 - 5 *Đóng kết nối giữa master node với cluster*
-

Ví dụ 3.3 trình bày một ứng dụng phân tán đơn giản tính biểu thức $a+b-c$. Đầu tiên, chúng ta định nghĩa phương thức *calculate()* để tính biểu thức trên. Trong hàm *main()*, gọi hàm *read_config()* để đọc thông tin cấu hình của cụm máy tính. Biến *hosts* chứa tên các máy tính làm worker node, biến *ports* chứa các ports mà các worker node lắng nghe các kết nối dựa trên giao thức TCP/IP. Sau đó, chúng ta tạo đối tượng *cluster* của lớp *Cluster* dựa vào danh sách worker nodes và ports. Đối tượng *cluster* này đại diện cho cluster xử lý phân tán. Kế tiếp, gọi phương thức *cluster.connect()* để kết nối master node với cluster. Để tiến hành xử lý phân tán, chúng ta gọi phương thức *cluster.run_method(calculate, a=10, b=8, c=2)* để gửi phương thức *calculate()* và các tham số $a=10$, $b=8$, $c=2$ đến các worker node trong cluster để thực thi. Kết quả trả về từ các worker nodes trong cluster được chứa trong biến *results*. Cuối cùng, gọi phương thức *cluster.close()* để đóng kết nối giữa master node và cluster. Chương trình này khi chạy sẽ thực hiện phương thức

$calculate(a=10, b=8, c=2)$ trên cả 3 worker nodes nên kết quả sẽ là danh sách $[16, 16, 16]$.

Ví dụ 3.3. Một ứng dụng tính toán phân tán đơn giản

```
1  async def calculate ( self , a , b , c ):
2      return a + b - c
3
4  async def main():
5      hosts , ports = read_config()
6      cluster = Cluster(hosts, ports)
7      await cluster.connect()
8      results = await cluster.run_method(calculate, a=10, b=8, c=2)
9      print(f"Results: { results }") # Results: [16, 16, 16]
10     await cluster.close()
```

Dể thực thi phương thức $calculate()$ với những bộ tham số khác nhau trên các worker nodes, chúng ta sẽ thay dòng lệnh số 9 trong Ví dụ 3.3 bằng dòng lệnh sau:

```
results = await cluster.run_method(calculate,
dict(a=10, b=8, c=2),
dict(a=100, b=80, c=20),
dict(a=1000, b=800, c=200))
```

Dòng lệnh trên sẽ thực hiện phương thức $calculate(a=10, b=8, c=2)$ trên $computer1$, thực hiện phương thức $calculate(a=100, b=80, c=20)$ trên $computer2$, và thực hiện phương thức $calculate(a=1000, b=800, c=200)$ trên $computer3$. Kết quả sẽ là $[16, 160, 1600]$.

Nếu chúng ta thay dòng lệnh số 9 trong Ví dụ 3.3 bằng dòng lệnh sau:

```
results = await cluster.run_method(calculate,
dict(a=10, b=8, c=2),
dict(a=100, b=80, c=20))
```

Kết quả sẽ là $[16, 160]$. Ở dòng lệnh trên, chúng ta chỉ cung cấp hai bộ tham số (a, b, c) nên chỉ có hai worker nodes thực thi phương thức $calculate()$, do đó chỉ có hai kết quả trả về.

3.3.2 Triển khai ứng dụng huấn luyện DNN

Dể triển khai ứng dụng huấn luyện DNN phân tán trên DDLF, chúng ta thực hiện các bước như trong Thuật toán 5.

Thuật toán 5: Triển khai ứng dụng huấn luyện mô hình DL trên DDLF

- 1 *Đọc cấu hình của cluster*
 - 2 *Tạo đối tượng của lớp Cluster*
 - 3 *Kết nối master node với cluster*
 - 4 *Tải dataset*
 - 5 *Tạo mô hình*
 - 6 *Huấn luyện mô hình*
 - 7 *Đóng kết nối giữa master node với cluster*
 - 8 *Dánh giá mô hình*
 - 9 *Lưu mô hình*
-

Dể thuận tiện khi triển khai một ứng dụng phân tán, chúng tôi đã tạo interface *IApp* để đặc tả các chức năng cần thiết. Khi triển khai ứng dụng, người phát triển ứng dụng chỉ cần tạo lớp *App* (có thể đặt tên khác theo ý muốn) hiện thực interface *IApp*, và override một vài phương thức cần thiết. Bảng 3.6 mô tả các phương thức được đặc tả trong interface *IApp*.

Bảng 3.6. Các phương được đặc tả trong interface *IApp*

#	Phương Thức	Mô Tả
1.	async def connect(self)	Tạo kết nối giữa master node với cluster.
2.	async def close(self)	Đóng kết nối giữa master node với cluster.
3.	async def load_dataset(self)	Tải dataset.
4.	async def create_model(self)	Tạo mô hình.
5.	async def evaluate_model(self)	Dánh giá mô hình.
6.	async def save_model(self, path)	Lưu mô hình vào file.
7.	async def load_model(self, path)	Tải mô hình từ file.
8.	async def train(self, weights, worker_epochs, batch_size)	Huấn luyện mô hình ở các worker nodes.

#	Phương Thức	Mô Tả
9.	async train_sync(self, master_epochs, worker_epochs, batch_size)	def Huấn luyện mô hình kiểu đồng bộ ở master node.
10.	async train_async(self, master_epochs, worker_epochs, batch_size)	def Huấn luyện mô hình kiểu bất đồng bộ ở master node.
11.	async def shut- down(self)	Shutdown cluster.

Ví dụ 3.4 trình bày một ứng dụng huấn luyện DNN phân tán kiểu đồng bộ trên DDLF. Đầu tiên, thông tin cấu hình của cụm máy tính được đọc để tạo đối tượng của lớp *App*. Sau đó, gọi phương thức *app.connect()* để tạo kết nối giữa master node và cluster. Phương thức *app.load_dataset()* được gọi để tải dataset về master node và các worker nodes. Tùy vào nhu cầu thực tế, chúng ta có thể tải toàn bộ dataset hoặc một partition của dataset về mỗi worker nodes. Gọi phương thức *app.create_model()* để tạo mô hình DL trên master node và worker nodes. Để huấn luyện mô hình, chúng ta gọi phương thức *app.train_sync()* cho kiểu đồng bộ hoặc *app.train_async()* cho kiểu bất đồng bộ. Tham số *master_epochs* dùng để khai báo số *epochs* cần thiết để huấn luyện mô hình trên master node. Tham số *worker_epochs* dùng để khai báo số *epochs* mà mô hình được huấn luyện trên mỗi worker node. Sau khi thực hiện đủ số *worker_epochs*, các worker nodes sẽ gửi *gradients* về master node để cập nhật mô hình chính ở master node. Cần lưu ý rằng, tham số *master_epochs* nên là bội số của tham số *worker_epochs*. Sau khi hoàn thành việc huấn luyện mô hình, chúng ta có thể đánh giá mô hình bằng phương thức *app.evaluate_model()*, và lưu mô hình để dùng lại sau này bằng phương thức *app.save_model(path)*. Cuối cùng, gọi phương thức *app.close()* để đóng kết nối giữa master node và cluster.

Ví dụ 3.4. Một ứng dụng huấn luyện DNN phân tán kiểu đồng bộ trên DDLF

```

1  async def main():
2      hosts, ports = read_config()
3      app = App(Cluster(hosts, ports))
4      await app.connect()
5      await app.load_dataset()
6      await app.create_model()
7      await app.train_sync(master_epochs=100, worker_epochs=10, batch_size=32)
8      await app.evaluate_model()
9      await app.save_model(path)
10     await app.close()

```

Ví dụ 3.5 trình bày việc hiện thực sơ lược của lớp *App*. Lớp *App* phải thừa kế từ interface *IApp* để dùng lại những phương thức đã được hiện thực trong interface *IApp*.

Do ứng dụng phải tương tác với cluster nên bên trong lớp *App* phải lưu tham chiếu của đối tượng lớp *Cluster*. Các phương thức *load_dataset()* và *create_model()* bắt buộc phải hiện thực theo yêu cầu cụ thể. Các phương thức *train()*, *train_sync()*, và *train_async()*, nếu chấp nhận sự hiện thực mặc định, chúng ta không cần override chúng; ngược lại, chúng ta có thể override chúng một cách thích hợp.

Ví dụ 3.5. Hiện thực lớp App

```
1 class App(IApp):
2     def __init__(self, cluster: Cluster):
3         super().__init__(cluster)
4
5     @async def load_dataset(self):
6         # your code here
7
8     @async def create_model(self):
9         # your code here
10
11    @async def train(self, worker_epochs, batch_size):
12        # your code here
13
14    @async def train_sync(self, master_epochs, worker_epochs, batch_size):
15        # your code here
16
17    @async def train_async(self, master_epochs, worker_epochs, batch_size):
18        # your code here
```

Ví dụ 3.6 trình bày chi tiết hiện thực phương thức *load_dataset()* để tải dataset MNIST.

Ví dụ 3.6. Phương thức *load_dataset()* của lớp *App*

```
1     @async def load_dataset(self):
2         # load the MNIST dataset
3         (self.x_train, self.y_train), (self.x_test, self.y_test) = datasets.mnist.load_data()
4
5         # Make sure images have shape (28, 28, 1)
6         self.x_train = np.expand_dims(self.x_train, -1)
7         self.x_test = np.expand_dims(self.x_test, -1)
8
9         # Normalize pixel values from [0, 255] to [-0.5, 0.5] to make it easier to work
10        with
11            self.x_train, self.x_test = (self.x_train / 255.0) - 0.5, (self.x_test / 255.0) - 0.5
```

Ví dụ 3.7 trình bày chi tiết hiện thực phương thức *create_model()* để tạo mô hình CNN cho việc phân loại trên dataset MNIST.

Ví dụ 3.7. Phương thức *create_model()* của lớp *App*

```
1     @async def create_model(self):
```

```
2     input_shape = (28, 28, 1) # for MNIST
3     # input_shape = (32, 32, 3) # for CIFAR10
4     # create the model
5     self.model = models.Sequential()
6     self.model.add(layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
7         input_shape=input_shape))
8     self.model.add(layers.MaxPooling2D(pool_size=(2, 2)))
9     self.model.add(layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
10    self.model.add(layers.Flatten())
11    self.model.add(layers.Dense(64, activation='relu'))
12    self.model.add(layers.Dense(10, activation='softmax'))
```

3.4 Đặc điểm của DDLF

DDLF có các ưu điểm sau đây:

1. Đơn giản: Mục đích thiết kế là giữ cho DDLF càng đơn giản càng tốt. Đơn giản khi cài đặt và sử dụng giúp dễ dàng tiếp cận và triển khai nhanh ứng dụng, dẫn đến tiết kiệm thời gian và công sức, cuối cùng đạt được mục tiêu hiệu quả về kinh tế. Để cài đặt DDLF trên cluster chỉ cần chạy script đơn giản. Để triển khai ứng dụng huấn luyện DNN phân tán chỉ cần mở rộng interface *IApp* là có thể sử dụng lại các phương thức tiện ích sẵn có, không cần lập trình phức tạp.
2. Da nhiệm hiệu quả: DDLF dùng package *asyncio* của *Python 3.7+* để thực hiện đa nhiệm hiệu quả hơn *multi-threading*, giúp hiệu quả xử lý tốt hơn.
3. Da nền tảng: DDLF có thể cài đặt trên các hệ điều hành khác nhau, chỉ cần hệ điều hành hỗ trợ *Python 3.7+*.
4. Linh hoạt: DDLF cho phép master node điều khiển các worker nodes mà không có bất kỳ hạn chế nào, tạo điều kiện thuận lợi để giải phóng tư duy lập trình, việc lập trình không bị phụ thuộc hay bị gò bó vào bất kỳ cấu trúc lập trình cứng ngắc nào.
5. Khả năng mở rộng: DDLF cho phép master node thêm động biến và phương thức vào các worker nodes để phục vụ nhu cầu thực tế đa dạng. Các biến và phương thức bổ sung tồn tại lâu dài trên worker nodes cho đến khi thực hiện thao tác xóa hoặc shutdown cluster. Điều này cho phép sử dụng lại biến và phương thức, giảm thiểu nhu cầu truyền dữ liệu hoặc chương trình từ master node đến worker nodes, dẫn đến rút ngắn thời gian xử lý. Đặc điểm này cũng giúp cho người lập trình tổ chức động dữ liệu và chương trình ở các worker nodes một cách dễ dàng và hợp lý, từ đó việc lập trình phân tán sẽ tự nhiên hơn, không gò bó trong khuôn khổ, và hiệu suất được nâng cao.
6. Hiệu quả: Qua thực nghiệm, DDLF chứng tỏ có hiệu quả tốt vì nó giúp huấn luyện nhanh DNN và tạo ra mô hình chất lượng.

DDLF cũng có những nhược điểm sau đây:

1. Chưa hỗ trợ đa ngôn ngữ: Hiện tại DDLF chỉ hỗ trợ ngôn ngữ Python.
2. Chức năng tiện ích chưa phong phú: Việc cung cấp nhiều chức năng tiện ích giúp phát triển nhanh ứng dụng. Tuy nhiên, trong phiên bản này chức năng tiện ích chưa nhiều, cần bổ sung thêm.
3. Tối ưu chi phí liên lạc: Trong quá trình huấn luyện DNN phân tán, việc liên lạc giữa master node và các worker nodes diễn ra thường xuyên và thường phải truyền số lượng lớn dữ liệu như weights/gradients, data. Vì vậy, một nền tảng tốt phải có biện pháp tối ưu hóa chi phí liên lạc. Tuy nhiên, trong phiên bản này DDLF chưa hiện thực các biện pháp tối ưu chi phí liên lạc.
4. Phương pháp huấn luyện chưa đa dạng: Hiện tại, DDLF chỉ hỗ trợ phương pháp huấn luyện như: song song hóa dữ liệu đồng bộ và bất đồng bộ.

3.5 Những hạn chế của Apache Spark và giải pháp trên DDLF

3.5.1 Vấn đề tổ chức động dữ liệu và chức năng ở worker nodes

Trên DDLF, người dùng có thể thêm/bớt biến hay phương thức vào worker nodes mà không có bất kỳ sự ràng buộc nào. Điều này giúp cho người dùng phát triển và triển khai ứng dụng phân tán dễ dàng. Trong khi đó, Apache Spark cho phép thực hiện việc này một cách hạn chế thông qua cấu trúc *map/reduce* và *shared variables*.

A. Vấn đề của cấu trúc *map/reduce*

Cấu trúc *map/reduce* của Apache Spark được áp dụng rất hiệu quả trong xử lý dữ liệu lớn nhưng không phù hợp trong việc huấn luyện DNN phân tán vì việc lập trình bị gò bó trong cấu trúc *map/reduce*, ràng buộc tư duy lập trình, dẫn đến việc triển khai ứng dụng huấn luyện DNN kém linh hoạt và không hiệu quả. Mặt khác, do phụ thuộc vào cấu trúc *map/reduce*, hàm *worker_train(data_iterator)* chỉ nhận một tham số *data_iterator* (là 1 data partition). Người lập trình không thể truyền thông tin khác cho hàm *worker_train()* khi cần. Dùng biện pháp thay thế thì quá phiền phức. Điều này làm khó triển khai ứng dụng.

Dể giải quyết vấn đề này, chúng tôi cho phép master node tự do gửi code đến các worker nodes để thực thi. Nội dung code có thể là một đoạn code hoặc một method. Ngoài ra, chúng tôi còn cho phép master node thêm động các method vào worker nodes. Các method này sẽ tồn tại trên các worker nodes cho đến khi master node yêu cầu xóa các methods hoặc shutdown cluster. Trong tương lai, chúng tôi sẽ cho phép các phương thức bổ sung này có thể tồn tại vĩnh viễn ở các worker nodes kể cả sau khi shutdown cluster. Giải pháp này mang đến những tiện ích sau:

- Code gửi đến các worker nodes không phụ thuộc vào bất kỳ cấu trúc lập trình nào (như *map/reduce*). Các method có thể có số lượng tham số và kiểu dữ liệu tham số tùy ý. Điều này giúp giải phóng tư duy lập trình, tạo điều kiện thuận lợi để phát triển ý tưởng.

- Đối với những phương thức được sử dụng lặp đi lặp lại nhiều lần, chúng ta không cần gửi chúng đến các worker nodes trước mỗi lần sử dụng như trong *map/reduce* mà chỉ cần gửi chúng đến các worker nodes một lần, sau đó sử dụng lại nhiều lần. Điều này giúp giảm lưu lượng truyền tải trên mạng.

B. Shared variables

Trong Apache Spark, cơ chế chia sẻ dữ liệu giữa master node và worker nodes rất hạn chế thông qua *broadcast* và *accumulated variables*. Broadcast variables là biến chỉ đọc, accumulated variables là biến chỉ cộng. Hơn nữa, các biến này có nội dung giống nhau trên tất cả các worker nodes và chúng sẽ bị mất khi kết thúc một Spark session.

Để việc chia sẻ dữ liệu được thuận tiện hơn, trên DDLF, chúng tôi cho phép master node có thể thêm động biến vào các worker nodes và thao tác thoải mái trên chúng. Nội dung các biến này có thể giống nhau hoặc khác nhau trên các worker nodes. Giống như các phương thức bổ sung, các biến này sẽ tồn tại lâu dài trên các worker nodes cho đến khi master node yêu cầu xóa chúng hoặc shutdown cluster. Ngoài ra, master node cũng có thể yêu cầu worker nodes truy cập hệ thống file cục bộ hoặc phân tán để tải hoặc lưu trữ dữ liệu ở các worker nodes. Giải pháp này có những ích lợi sau:

- Thuận tiện trong việc tổ chức dữ liệu ở các worker nodes, tạo điều kiện thuận lợi để phát triển ý tưởng và triển khai các thuật toán.
- Do các biến được thêm động có thể tồn tại cho đến khi shutdown cluster nên master node chỉ cần gửi dữ liệu đến worker nodes một lần rồi sử dụng lại nhiều lần, thậm chí qua các ứng dụng khác nhau. Điều này giúp giảm chi phí liên lạc giữa master node và các worker nodes.

3.5.2 Việc xử lý dữ liệu huấn luyện

Trong Thuật toán 2 và Thuật toán 3, việc xáo trộn dữ liệu, phân hoạch lại dữ liệu và gửi các phân hoạch mới đến các worker nodes trong mỗi lần lặp là không thể tránh khỏi. Nhưng đây lại là những thao tác tốn nhiều thời gian xử lý, làm ảnh hưởng đến hiệu quả của toàn bộ hệ thống huấn luyện DNN phân tán. Chúng tôi đề xuất giải pháp như sau:

- Worker nodes sẽ tải sẵn toàn bộ dữ liệu huấn luyện nếu trong cache chưa có.
- Ở mỗi lần lặp, master node chỉ cần xáo trộn chỉ số của tập dữ liệu huấn luyện, phân hoạch tập chỉ số, và gửi các phân hoạch chỉ số đến worker nodes.
- Worker nodes sẽ lấy phân hoạch dữ liệu huấn luyện của riêng nó dựa vào tập dữ liệu huấn luyện đã tải sẵn trước đó và phân hoạch chỉ số đã nhận.

Giải pháp này có các ưu điểm sau:

- Giảm tối đa kích thước dữ liệu được gửi từ master node đến các worker nodes vì kích thước của 1 chỉ số, là một số nguyên, rất nhỏ so với kích thước của 1 dữ liệu huấn luyện, nhất là dữ liệu ảnh. Điều này dẫn đến giảm tối đa thời gian gửi dữ liệu, giúp cải thiện đáng kể hiệu suất của hệ thống huấn luyện DNN phân tán.

- Các datasets có thể được cache trong bộ nhớ hoặc hệ thống file cục bộ của worker nodes để sử dụng lại trong các ứng dụng huấn luyện DNN khác nhau.
- Tránh tạo một điểm thắt cổ chai tại master node do master node không phải phân phối khối lượng lớn dữ liệu huấn luyện cho tất cả worker nodes.

Thuật toán 6 là phiên bản cải tiến của Thuật toán 2 dựa vào giải pháp trên. Thuật toán 3 cũng có thể được cải tiến tương tự.

Thuật toán 6: Huấn luyện phân tán DNN kiểu đồng bộ trên K worker nodes được cải tiến

```

function master_train()
1   dataset ← load_dataset()
2   indices ← get_indices(dataset)
3   model ← create_model()
4   broadcast_model(model)
5   w ← model.get_weights()
6   i ← 0
7   while  $i < M$  do
8     indices ← shuffle(indices)
9      $\cup_{j=1}^K I_j$  ← partition(indices)
10    send_indices( $\cup_{j=1}^K I_j$ )
11    broadcast_weights(w)
12    g ← await collect( $g_j$ ) from all workers
13     $w \leftarrow w - \frac{1}{K} \times \sum_{j=1}^K g_j$ 
14    model.set_weights(w)
15    if is_convergent(model) then
16      broadcast_signal(STOP)
17      break
18    i ← i + N

```

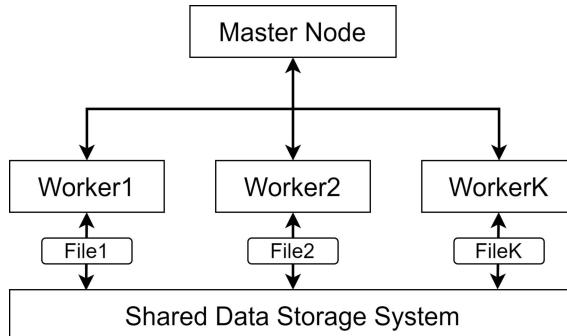
```

function workerj_train()
1   dataset ← load_dataset()
2   modelj ← await receive_model()
3   i ← 0
4   while  $i < M$  do
5      $I_j$  ← await receive_indices()
6      $P_j$  ← get_partition(dataset,  $I_j$ )
7      $w_j$  ← await receive_weights()
8     modelj.set_weights( $w_j$ )
9     modelj.fit( $P_j$ , epochs =  $N$ , batch_size =  $BS$ )
10     $g_j$  ← modelj.get_gradients()
11    send_gradients( $g_j$ ) to the master node
12    i ← i + N
13    if has_signal(STOP) then
14      break

```

Giải pháp trên giải quyết tốt cho trường hợp toàn bộ tập dữ liệu huấn luyện có thể lưu trữ vừa trong một worker node. Tuy nhiên, khi kích thước của tập dữ liệu huấn luyện quá lớn, vượt quá khả năng lưu trữ và xử lý trên một worker node; chúng ta cần cải tiến giải pháp trên như Hình 3.2. Đầu tiên, chúng ta có thể phân hoạch tập dữ liệu huấn luyện thành K files, với K là số lượng worker nodes. Chúng ta lưu các file dữ liệu này trong bất

kỳ hệ thống lưu trữ dùng chung nào mà tất cả worker nodes có thể truy cập được như: HDFS, S3, Cloud Storage. Sau đó, mỗi worker node sẽ tải file dữ liệu dành riêng cho nó để thực hiện việc huấn luyện DNN nếu trong cache chưa có dữ liệu. Sau khi tải, các file dữ liệu này có thể được cache trong bộ nhớ hoặc hệ thống file cục bộ của worker nodes để sử dụng lại. Giải pháp này rất hữu ích khi một tập dữ liệu được dùng cho nhiều ứng dụng huấn luyện DNN khác nhau, hoặc khi một DNN cần phải huấn luyện nhiều lần; vì không phải tải lại một khối lượng lớn dữ liệu.



Hình 3.2. Tải các datasets vào DDLF

Những giải pháp trên được hiện thực dễ dàng trên DDLF nhờ vào tính linh hoạt của nó (như cho phép thêm động biến và phương thức vào worker nodes, cho phép truy cập hệ thống file cục bộ của worker nodes). Tuy nhiên, trên Apache Spark, chúng ta rất khó triển khai các giải pháp này do Apache Spark không cho phép master node truy cập vào bộ nhớ và hệ thống file của worker nodes một cách linh hoạt.

3.5.3 Vấn đề liên lạc bất đồng bộ giữa master node và các worker nodes

Apache Spark không hỗ trợ liên lạc bất đồng bộ giữa master node và các worker nodes. Do đó, để triển khai ứng dụng huấn luyện DNN kiểu bất đồng bộ, người phát triển ứng dụng phải tự bổ sung cơ chế liên lạc bất đồng bộ vào Apache Spark hoặc sử dụng một framework có hỗ trợ cơ chế liên lạc bất đồng bộ.

Trên DDLF, master node tự do điều khiển các hoạt động ở worker node mà không có bất kỳ sự hạn chế nào. Vì vậy, khi cần, master node có thể liên lạc với worker nodes theo kiểu đồng bộ hoặc bất đồng bộ rất tự nhiên.

3.6 Thách thức trong huấn luyện DNN phân tán

Một thách thức lớn trong huấn luyện DNN phân tán là việc truyền dữ liệu giữa master node và worker nodes. Việc truyền một khối lượng dữ liệu lớn (data, gradients) thường xuyên sẽ làm chậm toàn bộ quá trình tính toán, ngay cả khi sử dụng mạng tốc độ cao.

Cụ thể, trong song song dữ liệu, các partition dữ liệu được gửi cho nhiều worker nodes. Mỗi worker node sẽ huấn luyện mô hình của mình bằng partition dữ liệu đã nhận để rút ra gradients. Các gradients này được gửi về master node để tính toán bộ lại bộ trọng số. Sau đó bộ trọng số mới lại được gửi đến tất cả worker nodes để cập nhật mô hình của

chúng. Quá trình này có thể rất chậm vì số lượng gradients bằng với số lượng trọng số. Mạng nơ-ron sâu thường chứa hàng triệu, thậm chí hàng tỷ tham số với độ chính xác 32-bit cần được trao đổi trong mỗi bước cập nhật.

Có hai cách để giảm chi phí liên lạc trong quá trình huấn luyện phân tán: 1) *Tăng kích thước minibatch (do giảm tần suất cập nhật)*; 2) *Giảm kích thước dữ liệu được trao đổi giữa các nút trong cụm tính toán*. Nhằm đạt được mục đích thứ hai, chúng tôi đã khảo sát các phương thức nén dữ liệu phổ biến hiện nay để xem có thể áp dụng chúng trong quá trình truyền dữ liệu hay không. Ưu điểm của việc nén dữ liệu là làm giảm kích thước của dữ liệu, dẫn đến giảm thời gian truyền dữ liệu. Tuy nhiên, phải tốn thêm chi phí nén và giải nén. Bảng 3.7 trình bày kết quả khảo sát trên các phương thức nén phổ biến nhất hiện nay. Chúng tôi sử dụng mạng LAN với tốc độ tối đa $1000\ Mbps$. Dữ liệu đầu vào là bộ trọng số rút ra từ mô hình được tạo bằng phương thức *create_model()* trong Ví dụ 3.7. Kích thước khi chưa nén của bộ trọng số là $2.062\ MB$. Cột *Methods* liệt kê các phương pháp nén dữ liệu, trong đó *None* có nghĩa là không nén. Cột *Data Size* liệt kê kích thước của bộ trọng số sau khi nén. Cột *Ratio* trình bày tỷ lệ nén. Ba cột kế tiếp trình bày thời gian nén, giải nén, truyền dữ liệu giữa master node và một worker node. Cột *Total* trình bày tổng thời gian xử lý $Total = Compression + Transmission + Decompression$.

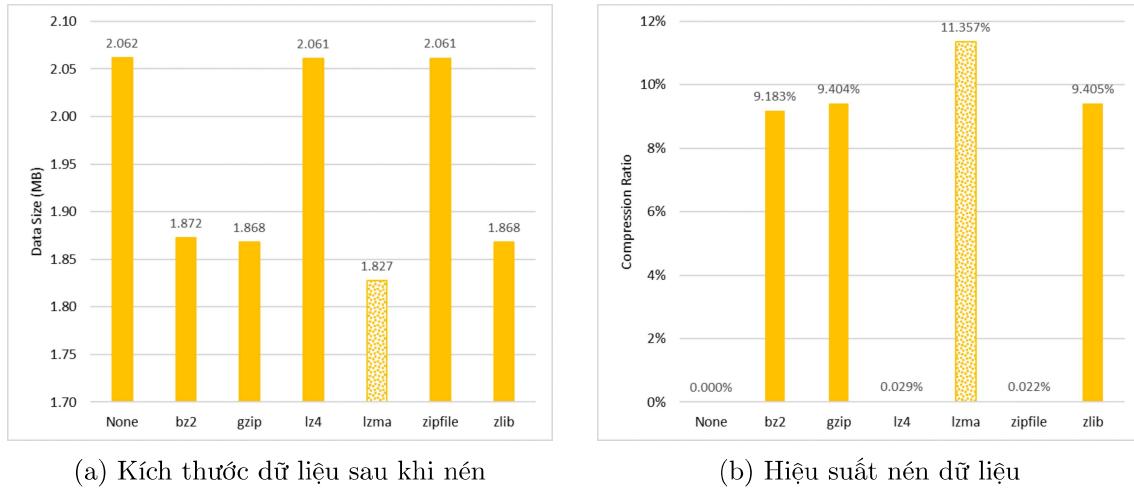
Bảng 3.7. So sánh các phương pháp nén dữ liệu

Methods	Data Size (MB)	Ratio	Time (Seconds)			
			Compression	Decompression	Transmission	Total
None	2.062	0.000%	0.000	0.000	0.438	0.438
bz2	1.872	9.183%	0.181	0.100	0.398	0.680
gzip	1.868	9.404%	0.092	0.012	0.397	0.501
lz4	2.061	0.029%	0.005	0.004	0.438	0.447
lzma	1.827	11.357%	0.418	0.095	0.389	0.902
zipfile	2.061	0.022%	0.003	0.004	0.438	0.445
zlib	1.868	9.405%	0.092	0.010	0.397	0.500

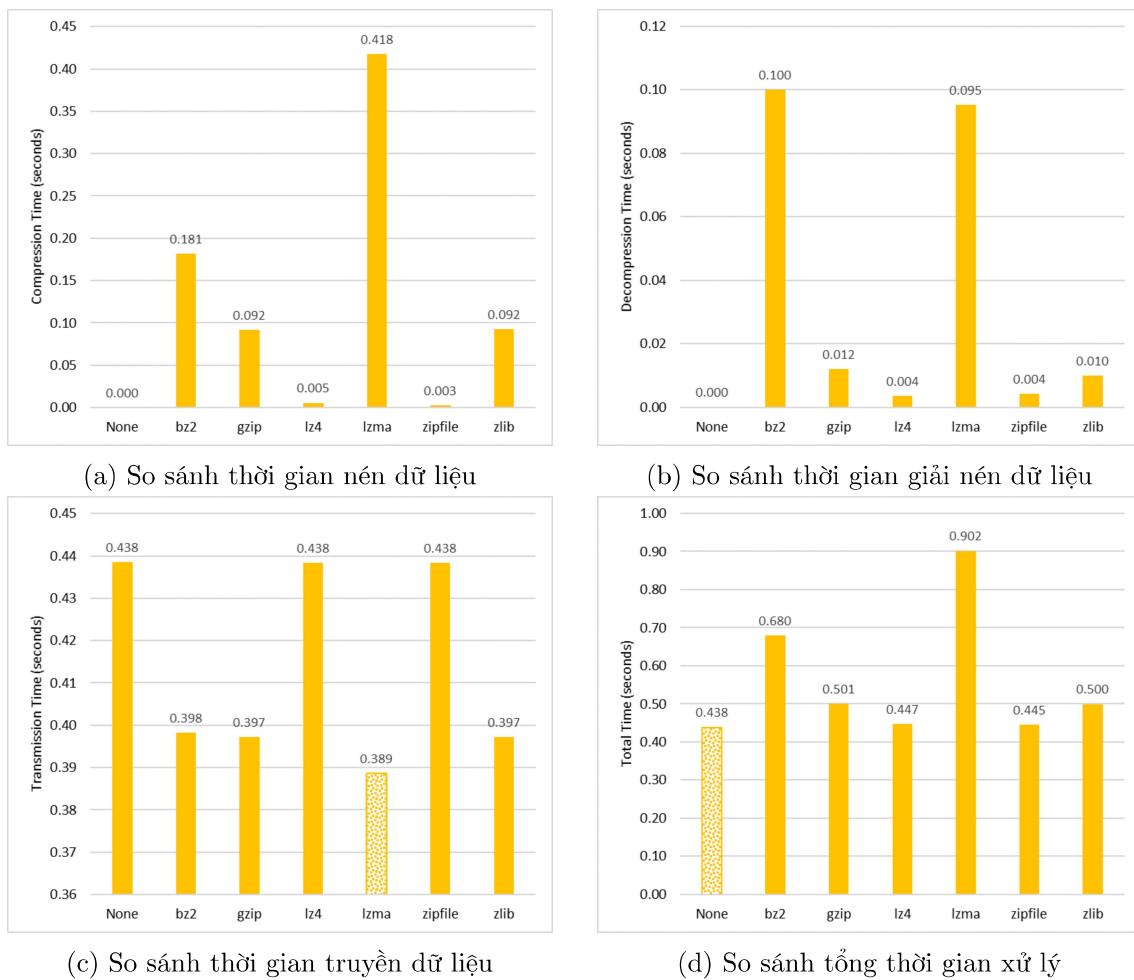
Hình 3.3 so sánh hiệu suất giữa các phương pháp nén dữ liệu, và cho thấy rằng phương pháp *lzma* đạt hiệu suất nén tốt nhất (11.357%).

Hình 3.4 so sánh thời gian xử lý khi áp dụng các phương pháp nén dữ liệu. Kết quả cho thấy, không áp dụng phương pháp nén dữ liệu đạt tổng thời gian xử lý nhanh nhất (0.438 giây). Phương pháp *lzma* tuy đạt hiệu suất nén tốt nhất nhưng thời gian nén lâu nhất (0.418 giây), thời gian giải nén cũng lâu thứ nhì (0.095 giây), dẫn đến tổng thời gian xử lý lâu nhất (0.902 giây).

Chúng tôi cũng thực nghiệm trên nhiều bộ dữ liệu được tạo ngẫu nhiên nhưng kết quả cũng không như kỳ vọng. Do đó, trong phiên bản hiện tại của DDLF, chúng tôi quyết định không nén dữ liệu khi truyền mà tiếp tục nghiên cứu thêm.



Hình 3.3. So sánh hiệu suất giữa các phương pháp nén dữ liệu



Hình 3.4. So sánh thời gian xử lý khi áp dụng các phương pháp nén dữ liệu

Chương 4

THỰC NGHIỆM & THẢO LUẬN

4.1 Datasets

Trong nghiên cứu này, chúng tôi sử dụng hai tập dữ liệu MNIST và CIFAR-10.

Tập dữ liệu MNIST (<http://yann.lecun.com/exdb/mnist/>) gồm 70.000 ảnh trắng đen, thuộc 10 lớp, biểu diễn các chữ số viết tay kích thước 28x28. Tập dữ liệu huấn luyện có 60.000 ảnh và tập dữ liệu kiểm thử có 10.000 ảnh.

Tập dữ liệu CIFAR-10 (<https://www.cs.toronto.edu/~kriz/cifar.html>) bao gồm 60.000 ảnh màu 32x32, thuộc 10 lớp, với 6.000 ảnh cho mỗi lớp. Tập dữ liệu huấn luyện có 50.000 ảnh và tập dữ liệu kiểm thử có 10.000 ảnh.

4.2 Cấu hình hệ thống thực nghiệm

Chúng tôi đã xây dựng cụm chín máy tính. Trong đó, một máy tính làm master node và tám máy tính làm worker nodes. Cấu hình chi tiết của cụm máy tính được liệt kê trong Bảng 4.1.

Bảng 4.1. Cấu hình cụm máy tính

#	Thuộc Tính	Master Node	Worker Nodes
1.	Processor	Intel(R) Core™ i5-6500 CPU @ 3.20GHz	Intel(R) Core™ i5-6500 CPU @ 3.20GHz
2.	Cores	8	4
3.	Thread(s) per core	1	1
4.	RAM	16.0 GB	8.0 GB

Chúng tôi đã tiến hành thực nghiệm trong hai môi trường: Spark Cluster và DDLF Cluster. Spark Cluster là cụm máy tính mà mỗi máy trong cụm được cài đặt nền tảng xử lý phân tán Apache Spark. Tương tự, DDLF Cluster là cụm máy tính mà mỗi máy trong cụm được cài đặt nền tảng huấn luyện DNN phân tán DDLF. Bảng 4.2 liệt kê chi

tiết các phần mềm được cài đặt trên cụm máy tính.

Bảng 4.2. Các phần mềm được cài đặt

#	Software	Spark Cluster	DDLF Cluster
1.	Java SDK 1.8.0 _ 201	✓	✓
2.	Python 3.8.5	✓	✓
3.	Apache Spark 3.0.0	✓	
4.	DDLF 2.0		✓

4.3 So sánh hiệu quả giữa Apache Spark với DDLF

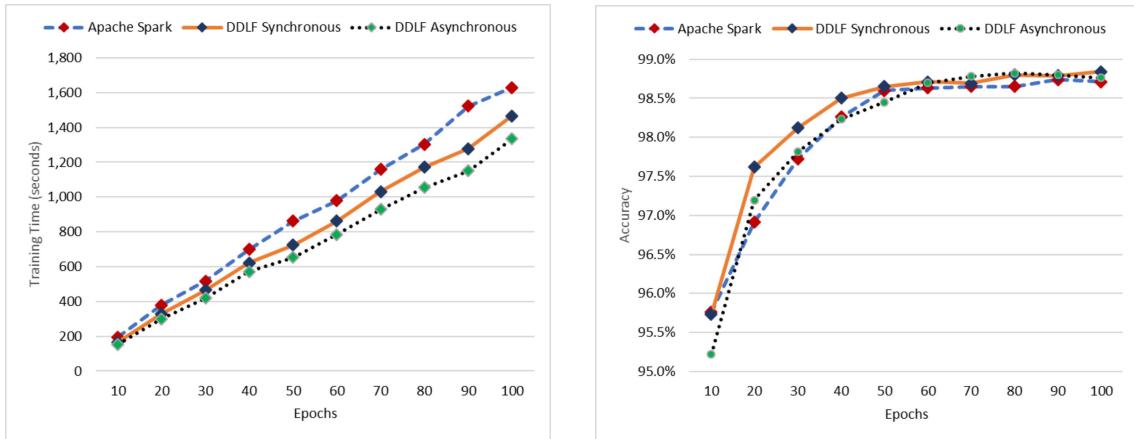
Để so sánh hiệu quả giữa Apache Spark và DDLF trong việc huấn luyện DNN phân tán, chúng tôi tiến hành ba loại thực nghiệm trên hai datasets MNIST và CIFAR10: 1) *Dùng Keras kết hợp với Apache Spark huấn luyện kiểu đồng bộ*; 2) *Dùng Keras kết hợp với DDLF huấn luyện kiểu đồng bộ*; 3) *Dùng Keras kết hợp với DDLF huấn luyện kiểu bất đồng bộ*. Mỗi loại thực nghiệm gồm có mươi thực nghiệm khác nhau dựa vào số lượng epochs thay đổi từ 10 đến 100, bước nhảy là 10. Mô hình CNN được tạo ra như trong Ví dụ 3.7. Chúng tôi không thực nghiệm trường hợp dùng Keras kết hợp với Apache Spark để huấn luyện kiểu bất đồng bộ do Apache Spark không hỗ trợ liên lạc bất đồng bộ giữa master node và các worker nodes như đã đề cập trong .

Bảng 4.3 trình bày chi tiết kết quả thực nghiệm trên MNIST. Hình 4.1 trình bày biểu đồ minh họa kết quả thực nghiệm. Biểu đồ trong Hình 4.1a cho thấy DDLF hiệu quả hơn Apache Spark vì thời gian huấn luyện nhanh hơn. Mặt khác, do huấn luyện kiểu bất đồng bộ tận dụng tối đa khả năng của các worker nodes nên hiệu quả hơn huấn luyện kiểu đồng bộ. Biểu đồ trong Hình 4.1b cho thấy độ chính xác của mô hình trong cả ba trường hợp chênh lệch không đáng kể.

Bảng 4.3. So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên MNIST

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
10	195.57	95.76%	166.23	95.73%	152.93	95.22%
20	378.69	96.91%	329.46	97.62%	299.81	97.20%
30	517.14	97.72%	465.43	98.12%	418.88	97.81%
40	698.14	98.26%	621.34	98.50%	571.64	98.23%
50	863.14	98.60%	725.04	98.65%	652.53	98.45%
60	979.79	98.63%	862.22	98.71%	784.62	98.70%
70	1,158.79	98.65%	1,031.33	98.69%	928.19	98.78%

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
80	1,302.70	98.65%	1,172.43	98.80%	1,055.19	98.82%
90	1,523.03	98.74%	1,279.34	98.79%	1,151.41	98.80%
100	1,630.16	98.71%	1,467.14	98.84%	1,335.10	98.76%



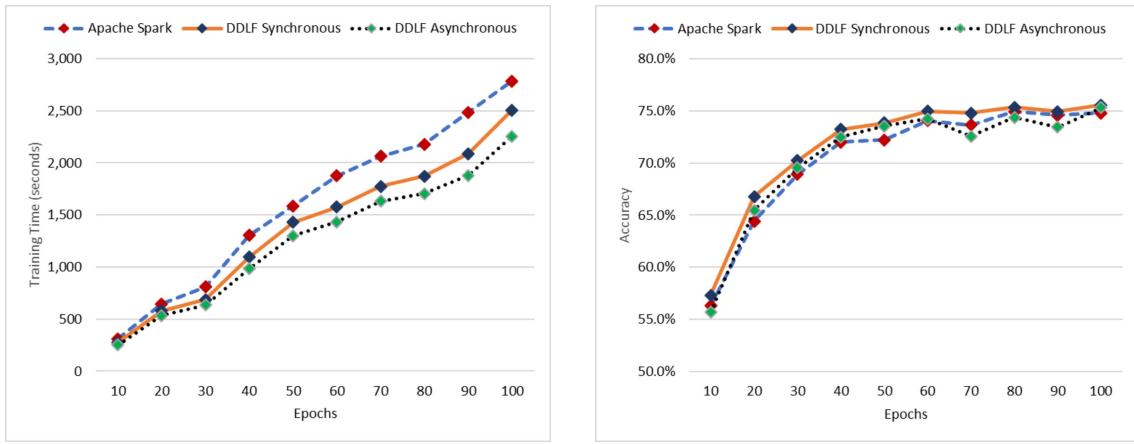
Hình 4.1. So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên MNIST

Bảng 4.4 trình bày chi tiết kết quả thực nghiệm trên CIFAR10. Hình 4.2 trình bày biểu đồ minh họa kết quả thực nghiệm. Tương tự như trên MNIST, kết quả thực nghiệm trên CIFAR10 cũng cho thấy DDLF hiệu quả hơn Apache Spark. Độ chính xác của mô hình trong cả ba trường hợp cũng gần nhau. Một khác, do CIFAR10 (chứa các ảnh màu) phức tạp hơn MNIST (chỉ chứa ảnh đen trắng) nên thời gian huấn luyện trên CIFAR10 lâu hơn trên MNIST khá nhiều.

Bảng 4.4. So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên CIFAR10

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
10	309.57	56.31%	275.52	57.31%	250.72	55.68%
20	644.34	64.42%	579.90	66.81%	533.51	65.48%
30	810.83	68.91%	689.21	70.26%	634.07	69.56%
40	1,303.50	72.01%	1,094.94	73.23%	985.45	72.49%
50	1,587.21	72.21%	1,428.49	73.86%	1,299.92	73.56%

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
60	1,876.13	74.09%	1,575.95	75.00%	1,434.12	74.25%
70	2,064.50	73.66%	1,775.47	74.82%	1,633.43	72.57%
80	2,179.49	74.96%	1,874.36	75.37%	1,705.67	74.38%
90	2,486.07	74.62%	2,088.30	74.95%	1,879.47	73.45%
100	2,785.08	74.81%	2,506.57	75.59%	2,255.91	75.34%



(a) So sánh thời gian huấn luyện

(b) So sánh độ chính xác của mô hình

Hình 4.2. So sánh thời gian huấn luyện và độ chính xác của mô hình được huấn luyện giữa Apache Spark và DDLF trên CIFAR10

Bảng 4.5 tổng kết kết quả thực nghiệm trên hai datasets MNIST và CIFAR10. Trung bình, DDLF đồng bộ nhanh hơn Apache Spark 12.80%, DDLF bất đồng bộ nhanh hơn Apache Spark 20.78% và DDLF bất đồng bộ nhanh hơn DDLF đồng bộ 9.15%.

Bảng 4.5. Tổng kết việc so sánh thời gian huấn luyện trên hai datasets

Mô Tả	Trung Bình Trên Dataset		Trung Bình
	MNIST	CIFAR10	
DDLF Synchronous nhanh hơn Apache Spark	12.40%	13.20%	12.80%
DDLF Asynchronous nhanh hơn Apache Spark	20.55%	21.01%	20.78%
DDLF Asynchronous nhanh hơn DDLF Synchronous	9.30%	9.00%	9.15%

Chương 5

KẾT LUẬN & HƯỚNG PHÁT TRIỀN

Trong kỷ nguyên dữ liệu lớn, các tập dữ liệu huấn luyện nhỏ ngày càng ít đi. Huấn luyện mô hình DL trong môi trường phân tán là mục tiêu phải hướng đến vì lợi ích thiết thực của nó. Apache Spark được xem là một nền tảng xử lý dữ liệu lớn nổi bật nhất hiện nay. Vì vậy, đã có nhiều nền tảng huấn luyện mô hình DL đã kết hợp với Apache Spark để tận dụng đặc điểm tính toán phân tán của nó. Tuy nhiên, mục đích thiết kế của Apache Spark là xử lý dữ liệu lớn chứ không nhằm để huấn luyện DNN. Việc áp dụng gượng ép Apache Spark vào việc huấn luyện phân tán DNN gặp nhiều trở ngại, dẫn đến giảm hiệu suất của hệ thống. Vì vậy, chúng tôi đã phát triển nền tảng xử lý phân tán DDLF, hoàn toàn độc lập với Apache Spark, khắc phục được những trở ngại mà chúng tôi gặp phải khi dùng Apache Spark để huấn luyện mô hình DL. Mặc dù mục tiêu ban đầu khi thiết kế DDLF là chuyên dùng huấn luyện phân tán mô hình DL, nhưng DDLF cũng có thể được dùng để triển khai các loại ứng dụng phân tán bất kỳ. Ưu điểm của nền tảng DDLF: 1) *Dơn giản, dễ triển khai ứng dụng phân tán;* 2) *Giúp lập trình tự nhiên, thoải mái;* 3) *Linh hoạt và có thể mở rộng;* 4) *Hiệu suất cao.* Bên cạnh đó, DDLF cũng có những nhược điểm sau: 1) *Hiện tại chỉ hỗ trợ ngôn ngữ lập trình Python và đòi hỏi version từ 3.7 trở đi do kỹ thuật bên dưới sử dụng package asyncio;* 2) *Các chức năng tiện ích chưa nhiều do mới được phát triển;* 3) *Chi phí liên lạc giữa master node và worker nodes chưa được tối ưu.*

Chúng tôi sẽ tiếp tục nghiên cứu và cải tiến DDLF như sau: 1) *Tích hợp một số component quan trọng viết bằng C/C++ để cải thiện hiệu suất;* 2) *Hỗ trợ thêm ngôn ngữ Scala để mở rộng cộng đồng sử dụng DDLF;* 3) *Bổ sung thêm nhiều chức năng tiện ích hỗ trợ việc huấn luyện phân tán;* 4) *Nghiên cứu tối ưu hóa chi phí liên lạc giữa master node và các worker nodes.*

TÀI LIỆU THAM KHẢO

- [1] M. Li **and others**, “Scaling Distributed Machine Learning with the Parameter Server,” *inProceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* **jourser** OSDI’14, Broomfield, CO: USENIX Association, 2014, 583–598, ISBN: 9781931971164.
- [2] M. Li, D. G. Andersen, A. J. Smola **and** K. Yu, “Communication Efficient Distributed Machine Learning with the Parameter Server,” *inAdvances in Neural Information Processing Systems* Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence **and** K. Q. Weinberger, **editors**, **volume** 27, Curran Associates, Inc., 2014. **url:** <https://proceedings.neurips.cc/paper/2014/file/1ff1de774005f8da13f42943881c655f-Paper.pdf>.
- [3] M. Abadi **and others**, *TensorFlow: A system for large-scale machine learning*, 2016. arXiv: 1605.08695 [cs.DC].
- [4] A. Harlap **and others**, *PipeDream: Fast and Efficient Pipeline Parallel DNN Training*, 2018. arXiv: 1806.03377 [cs.DC].
- [5] P. Moritz **and others**, *Ray: A Distributed Framework for Emerging AI Applications*, 2018. arXiv: 1712.05889 [cs.DC].
- [6] A. Wang, X. Jia, L. Jiang, J. Zhang, Y. Li **and** W. Lin, “Whale: A Unified Distributed Training Framework,” *CoRR*, **jourvol** abs/2011.09208, 2020. arXiv: 2011.09208. **url:** <https://arxiv.org/abs/2011.09208>.
- [7] R. Rajak, “Cluster Computing: Emerging Technologies, Benefits, Architecture, Tools and Applications,” *International Journal of Advanced Science and Technology*, **jourvol** 29, **number** 04, **pages** 4008 –, 2020. **url:** <http://sersc.org/journals/index.php/IJAST/article/view/24569>.
- [8] F. Es-Sabery **and** A. Hair, “Big Data Solutions Proposed for Cluster Computing Systems Challenges: A Survey,” *inProceedings of the 3rd International Conference on Networking, Information Systems & Security* **jourser** NISS2020, Marrakech, Morocco: Association for Computing Machinery, 2020, ISBN: 9781450376341. DOI: 10.1145/3386723.3387826. **url:** <https://doi.org/10.1145/3386723.3387826>.
- [9] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine **and** S. Khan, “A Survey of Distributed Data Stream Processing Frameworks,” *IEEE Access*, **jourvol** 7, **pages** 154 300–154 316, 2019.

- [10] E. Shaikh, I. Mohiuddin, Y. Alufaisan **and** I. Nahvi, “Apache Spark: A Big Data Processing Engine,” **in** *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM) 2019*, **pages** 1–6. DOI: 10.1109/MENACOMM46666. 2019.8988541.
- [11] L. Deng **and** D. Yu, “Deep Learning: Methods and Applications,” Microsoft, tech-report MSR-TR-2014-21, 2014. **url:** <https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/>.
- [12] D. Lei, X. Chen **and** J. Zhao, “Opening the black box of deep learning,” **may** 2018. arXiv: 1805.08355v1 [cs.LG].
- [13] D. T. Chang, “Concept-Oriented Deep Learning,” **june** 2018. arXiv: 1806.01756v1 [cs.AI].
- [14] M. Naumov **and others**, *Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems*, 2020. arXiv: 2003.09518 [cs.DC].
- [15] SparkTeam, *Apache Spark FAQ*, <https://spark.apache.org/faq.html>, accessed 6/10/2021, 2021.
- [16] O. G. Yalçın, *Top 5 Deep Learning Frameworks to Watch in 2021 and Why TensorFlow*, <https://towardsdatascience.com/top-5-deep-learning-frameworks-to-watch-in-2021-and-why-tensorflow-98d8d6667351>, accessed 6/8/2021.
- [17] M. Langer, Z. He, W. Rahayu **and** Y. Xue, “Distributed Training of Deep Learning Models: A Taxonomic Perspective,” *IEEE Transactions on Parallel and Distributed Systems*, **jourvol** 31, **number** 12, **pages** 2802–2818, 2020. DOI: 10.1109/TPDS.2020.3003307.
- [18] A. Sharma, “Guided parallelized stochastic gradient descent for delay compensation,” *CoRR*, **jourvol** abs/2101.07259, 2021. arXiv: 2101.07259. **url:** <https://arxiv.org/abs/2101.07259>.
- [19] K. S. Chahal, M. S. Grover, K. Dey **and** R. R. Shah, “A Hitchhiker’s Guide On Distributed Training Of Deep Neural Networks,” *Journal of Parallel and Distributed Computing*, **jourvol** 137, **pages** 65–76, 2020, ISSN: 0743-7315. DOI: <https://doi.org/10.1016/j.jpdc.2019.10.004>. **url:** <https://www.sciencedirect.com/science/article/pii/S0743731518308712>.

KẾT QUẢ ĐẠT ĐƯỢC

Một phần kết quả nghiên cứu được trình bày trong bài báo sau và đã được chấp thuận đăng trong tạp chí Intelligent Data Analysis, IOS Press:

- [1] Trung Phan, Phuc Do. “*A Novel Framework to Enhance the Performance of Training Distributed Deep Neural Networks*”. Intelligent Data Analysis, IOS Press, 2023.

Nội dung đính kèm sau đây bao gồm:

1. Nội dung đánh giá bài báo.
2. Mail chấp thuận đăng bài báo.
3. Nội dung bài báo.



Đỗ, Phúc <phucdo@uit.edu.vn>

Evaluation of your manuscript (M22-161-6710)

4 messages

Editor-IDA <editor@ida-ij.com>

Thu, May 5, 2022 at 8:50 PM

To: phucdo@uit.edu.vn

Cc: trungphanhsg@gmail.com, trung.phanhong@hoasen.edu.vn

Ref: M22-161-6710

Dear authors ,

We are pleased to let you know that your article entitled:

" A Novel Framework to Enhance the Performance of Training ..."

has been accepted for publication in the IDA journal. Please follow the comments provided by the two reviewers (given below this e-mail) and send your revised manuscript (in LaTex or Ms-Word) within the next 2-3 months. Please provide details of all the changes that you make in the revised version of your manuscript. If you need a LaTex style file, please let us know.

For the overall format of the paper, please refer to the IDA Journal Web Site given below (Instruction for Authors). For any additional information, please do not hesitate to contact us. Please make sure to include affiliations and e-mail addresses of all authors along with 3-5 Keywords on the title page of your revised manuscript.

*** Please note that authors of accepted articles are required to pay US\$350 or €300 publication fee before their papers being published. We will send you the payment instructions when we receive your revised manuscript.

Thank you for your interest in the IDA journal.

With kind regards,

Journal office

Intelligent Data Analysis - An International Journal

P.O. Box 46117, 2339 Ogilvie Rd.

Ottawa, ON K1J 9M7 Canada

<http://www.iospress.nl>

<http://www.iospress.nl/journal/intelligent-data-analysis>

Phone: 1(613) 355-8192

E-Mail: editor@ida-ij.com

=====

#1

IDA Journal Reviewer's Guidelines

=====

Paper Title: A Novel Framework to Enhance the Performance of Training
Distributed Deep Neural Networks

Name of First Author: T-H Phan

Name of Reviewer: xxxxxxxxxxxxxxxxxxxxxxx

IDA Reference Number: M22-161-6710

Section 1: Comments for Editor-in-Chief (optional)

Section 2: Comments for author(s)

Please answer the following questions:

1. Are the contents of the paper technically correct?

Yes.

2. Are the contents of the paper original?

Somewhat. The problem is well understood. The solution is not entirely original but it brings together different ideas in a way that creates a more efficient framework.

3. Are the paper clarity and presentation satisfactory?

Not entirely. The following should help the authors improve.

- There is a lot of redundancy throughout the paper where things are repeated unnecessarily in a number of places. The paper can and should be significantly shortened so the authors can make their point once and support it with the architecture and results. For example, section 3.3 is much too long and should be significantly shortened; the first bullet point in section 4.1 has already been stated earlier; section 4.6.1.A is a repeat of materials in sections 4.1 and 4.5; the beginning of section 4.6.2 is again a repeat of points made earlier; Table 2 is redundant and unnecessary (the text is sufficient).

- The authors state that one of the problems with current frameworks is that it is difficult to know the cause of failure since the framework is proprietary (section 4.2), and yet they propose their own, which presumably would cause the same issue for other users!! Only they would know how to use it, and thus they are creating for others the problem they are trying to solve for themselves. This deserves discussion/justification, or change.

- Section 4.6.2. Maybe I am missing something, but the architecture in Fig 5 still does not answer the issue if the dataset being too large to fit in memory, or does it? If so, please expand and explain; it still is unclear.

- Taking averages across MNIST and CIFAR for speed up as the authors do may not be appropriate. Furthermore, there is clearly a time element to the observations. For small number of epochs, there is very little observable difference in performance; as the number of epoch grows, the difference becomes more visible (and probably significant). The authors should discuss this, and maybe make the point that their framework is thus particularly helpful for DNN where training typically requires many epochs. In any case, they cannot just get an average speed up from their data; it would have to be done at each time step, or computed across the entire curves somehow.

4. Is the paper relevant to the Aims and Scope of IDA journal?

Yes.

Results:

Please classify this paper under one of the following four classes:

III. To be submitted after major revisions?

Your confidence: [X] High

Please provide comments if you classified the paper as: II, III or IV.

The paper has merit; the framework is interesting and seems to be more efficient than Spark. The authors must shorten the paper, address the comments above, and make their contribution concise and clear.

=====

#2

IDA Journal Reviewer's Guidelines

=====

Paper Title: A Novel Framework to Enhance the Performance of Training
Distributed Deep Neural Networks

Name of First Author: Trung Hong Phan

Name of Reviewer: xxxxxxxxxxxxxxxxxxxxxxxxx

IDA Reference Number: M22-161-6710

Section 1: Comments for Editor-in-Chief (optional)

Section 2: Comments for author(s)

Please answer the following questions:

1. Are the contents of the paper technically correct?

Yes it seems so

2. Are the contents of the paper original?

Yes quite reasonable and seems to be acceptable.

3. Are the paper clarity and presentation satisfactory?

There are some points that need to be worked out, listed below

4. Is the paper relevant to the Aims and Scope of IDA journal?

Yes it seems so

Results:

Please classify this paper under one of the following four classes:

- I. Acceptable without change?
- II. Acceptable with some minor revisions?
- =YES, I would consider that
- III. To be submitted after major revisions?
- IV. To be rejected?

Your confidence: High
 Medium
 Low

Please provide comments if you classified the paper as: II, III or IV.

- Well prepared manuscript that fits the aims and scope of the IDA journal.
- My suggestions are given below to be addressed.

Additional Comments (please provide details):

- Given the topic and the scope of work, it seems that there could be more research work to be covered in the Related works section and also some of the contributions could be quantified for their objectives.
- In both figures 4 and 5, would there be any role for domain knowledge, depending on the application domain? Please specify. or use of domain data?
- in section 5 (Experiments and data used), it would be helpful to explain certain important data characteristics, such as noise and missing values, that may exist in the data sets discussed and used in this study.
We are talking about the overall reliability of the data sets applied.
This information may have an effect on the entire approach and of course on the results of the study? Please explain.
- Basic editing of the text of the manuscript would be helpful to improve the text and its readability.

your accepted manuscript (Ref: 226710)

Editor-IDA <editor@ida-ij.com>

Mon, Jun 6, 2022, 8:21 PM

to phucdo, k.Look, me, trung.phanhong

Authors: Trung Phan and Phuc Do
Reference Number: 226710 (original: M22-161-6710)

Dear Phuc Do,

We have now evaluated the revised version of your manuscript and all the revisions were acceptable. Please note that your paper will appear in Volume 27(3) of the IDA journal. The publication date for this issue is April 2023.

The production department of IOS press, the publisher of the IDA journal, will contact you for proofreads in late March 2023. When you receive your proofreads please reply to the IOS Press production department within 48 hours.

More information is available at our website:
<http://www.iospress.nl/journal/intelligent-data-analysis/>.

We would like to thank you again for all your efforts and your interest in the IDA journal.

With best regards,

Journal office
Intelligent Data Analysis - An International Journal
P.O. Box 46117, 2339 Ogilvie Rd.
Ottawa, ON K1J 9M7 Canada
<http://www.iospress.nl>
<http://www.iospress.nl/journal/intelligent-data-analysis>
Phone: 1(613) 355-8192
E-Mail: editor@ida-ij.com

=====

A Novel Framework to Enhance the Performance of Training Distributed Deep Neural Networks

Trung Phan^{a,b}, Phuc Do^{a,*}

^a Faculty of Information Science And Engineering, University of Information Technology Vietnam National University, Ho Chi Minh City, Vietnam

E-mail: phucdo@uit.edu.vn

^b Faculty of Information Technology, Hoa Sen University, Ho Chi Minh City, Vietnam

E-mail: trung.phanhong@hoasen.edu.vn

Abstract. There are many attempts to implement deep neural network (DNN) distributed training frameworks. In these attempts, Apache Spark was used to develop the frameworks. Each framework has its advantages and disadvantages and needs further improvements. In the process of using Apache Spark to implement distributed training systems, we ran into some obstacles that significantly affect the performance of the systems and programming thinking. This is the reason why we developed our own distributed training framework, called Distributed Deep Learning Framework (DDLF), which is completely independent of Apache Spark. Our proposed framework can overcome the obstacles and is highly scalable. DDLF helps to develop applications that train DNN in a distributed environment (referred to as distributed training) in a simple, natural, and flexible way. In this paper, we will analyze the obstacles when implementing a distributed training system on Apache Spark and present solutions to overcome them in DDLF. We also present the features of DDLF and how to implement a distributed DNN training application on this framework. In addition, we conduct experiments by training a Convolutional Neural Network (CNN) model with datasets MNIST and CIFAR-10 in Apache Spark cluster and DDLF cluster to demonstrate the flexibility and effectiveness of DDLF.

Keywords: Distributed Deep Learning Framework, Distributed Neural Network, Distributed Processing System, Distributed Training, Apache Spark

1. Introduction

It takes a lot of time to train a DNN. The larger the training dataset, the better the model, but the longer the training time will be [1, 2]. Usually, the training process takes days or even weeks until convergence [3, 4]. We need to speed up this process by building a distributed training system that takes advantage of cluster computing capacity. Apache Spark is the most prominent and popular big data processing framework today. So many DNN training frameworks have been combined with Apache Spark to take advantage of its distributed computing feature. However, in the process of developing DNN training applications based on Apache Spark we encountered some obstacles [5–7]. Apache Spark's *map/reduce* structure is very useful when dealing with big data, but it has proved unsuitable for DNN training. The forced combination of Apache Spark for DNN training reduces the overall performance of the system.

* Corresponding author. E-mail: phucdo@uit.edu.vn.

As a result, we developed the DDLF distributed processing framework, which is completely independent of Apache Spark, to overcome the obstacles of training DNN on Apache Spark.

Our contributions can be summarized as:

- Proposing a new framework named DDLF for distributed DNN training.
- Describing the features of DDLF and how to implement a distributed DNN training application on DDLF.
- Analyzing the obstacles when implementing a distributed training system on Apache Spark and presenting solutions to overcome them in DDLF.
- Comparing the performance of DDLF and Apache Spark in distributed DNN training.

The rest of our paper is organized as follows: Section 2 is related works, Section 3 provides the basic concepts, Section 4 presents the methodology, Section 5 is about experiments, and the last section is the conclusions and future works.

2. Related works

Many research and development activities on distributed DNN training frameworks have been done as a result of the remarkable progress of deep learning (DL). Li et al. proposed a parameter server framework for distributed learning and several approaches were proposed to reduce the communication cost between nodes, such as only exchanging non-zero parameter values, local caching of index list, and randomly ignoring transmitted messages [8, 9]. Abadi et al. presented TensorFlow, a centralized framework for DNN training that integrates model and data parallelism [10]. Both works support asynchronous communication to improve efficiency but they do not control the stability of gradient updates.

Harlap et al. also proposed a distributed pipelined system for DNN training [11]. This work focuses on pipelining with a parallel model, partitioning DNN layers on different machines, and dividing the execution of machines by injecting consecutive mini-batches into the first one. This approach reduces communication load because only the operations and gradients of a subset of classes are communicated between machines. However, complex mechanisms (such as profiling, partitioning algorithms, and replicated stages) are needed to balance workloads between different machines, otherwise, computational resources will be wasted.

The next generation of machine learning (ML) applications will continuously interact with the environment and learn from these interactions. These applications place new and demanding system requirements, both in terms of performance and flexibility. Moritz et al. proposed Ray - a distributed system to solve them [12]. Ray implements a unified interface that can represent both task-parallel and actor-based computations, supported by a single dynamic execution engine. To meet performance requirements, Ray uses a distributed scheduler and a distributed fault-tolerant storage system to manage the control state of the system. In their experiments, they demonstrated scaling beyond 1.8 million tasks per second and better performance than existing dedicated systems for some challenging reinforcement learning applications.

With the growth in data size and model size, the data parallelism method cannot work on models whose parameter size cannot fit into the device memory of a single GPU. To further improve industrial-grade huge model training, Wang et al. introduced *Whale*, a unified distributed training framework [13]. It provides comprehensive parallel strategies including data parallelism, model parallelism, pipelines, hybrid strategies, and automatic parallel strategies. *Whale* is TensorFlow compatible and training tasks can be

easily distributed by adding a few lines of code without changing the user model code. According to the authors, *Whale* is the first work that can support different hybrid distributed strategies in a framework. In their experiment on the BERT (Bidirectional Encoder Representations from Transformers) large model, *Whale* pipeline strategy is 2.32x faster than Horovod data parallelism on 64 GPUs. In large-scale image classification tasks (100,000 classes), *Whale* hybrid strategy, which consists of operator sharding and data parallelism, is 14.8x faster than Horovod data parallelism on 64 GPUs.

3. Preliminaries

3.1. Computer cluster

A computer cluster is a group of computers connected to a network and they act like a single entity [14, 15]. There are many computer cluster architectures, Fig. 1 is a typical computer cluster. Each computer of a cluster is called a node. Where:

- Master node: is the computer that runs the main program, sends requests to the worker nodes to execute in parallel and collects the results.
- Worker node: is the computer involved in processing the requests of the master node.

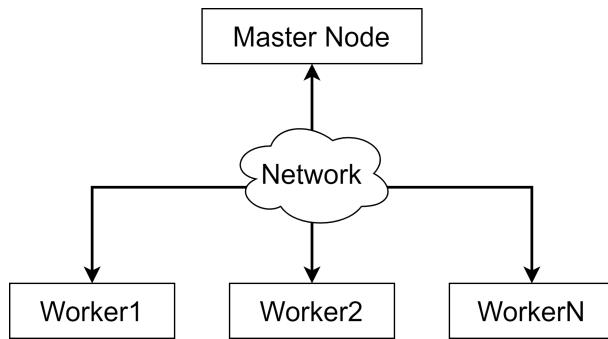


Fig. 1. A computer cluster

A computer cluster provides a distributed processing solution to solve complex problems by dividing a complex problem into many small parts, each of which is handled by a worker node. A computer cluster has many advantages such as cost-effectiveness, processing speed, high availability, scalability, and flexibility.

3.2. Spark cluster

There are many distributed processing frameworks such as Hadoop, Apache Spark, Apache Storm, Samza, Flink [16, 17]. Where, Apache Spark is the most powerful and widely used distributed processing framework for big data because of the following features: 1) *The processing speed is said to be lightning-fast*; 2) *Easy to use and flexible*; 3) *Providing support for complex analytics*; 4) *Real-time stream processing*.

To deploy a distributed processing system on the Apache Spark framework, a Spark cluster must be built. The Spark cluster is a computer cluster where each computer is installed with Apache Spark software to control the operation in the cluster. Because Apache Spark does not have its file management system, the Spark cluster is often combined with a distributed file system such as Hadoop's HDFS (Hadoop Distributed File System) [18] or Amazon's S3 [19].

3.3. DNN training

There are two ways to train DNN: local training and distributed training. If the training dataset is small (fits into the memory of a single computer) and the DNN is simple, we should use local training. On the contrary, we should choose distributed training. However, distributed training is more complex and incurs additional communication costs between machines in the processing cluster.

The accuracy of a DL model can increase with the number of training samples, the number of model parameters, or both. However, training large networks is complex and time-consuming when trained on a single machine, even with multithreading support. This requires training the DL models on many connected computers in a distributed manner.

3.3.1. Training DNN in a local environment

Currently, there are many frameworks for implementing DL. The five best ones are TensorFlow, Keras, PyTorch, Apache MXNet, and Microsoft Cognitive Toolkit [20]. Among them, Keras is the simplest and easiest to use. Keras's salient features include 1) *The API is easy to understand and consistent*; 2) *Keras can combine with many DL frameworks such as TensorFlow, Theano, and CNTK*; 3) *Keras supports parallel and distributed training on multiple GPUs*. Besides, TensorFlow is the best and most preferred framework today. Highlights of Tensorflow include: 1) *Powerful multi-GPU support*; 2) *Visualize calculation graphs*; 3) *Great documentation and community support*. However, Tensorflow is seen as a complex, difficult-to-use framework. Therefore, Keras was chosen as Tensorflow 2's high-level API for ease of access and efficiency when solving DL problems. From the above analysis, we choose Keras as the framework to train DNN in the local environment in this study.

3.3.2. Training DNN in a distributed environment

The accuracy of a DL model can be increased with the number of training samples, the number of model parameters, or both. However, training large models is complex and time-consuming when training in a local environment with a single computer. To speed up the DNN training, we train a DNN model in a distributed environment. There are two main ways of distributed training: model parallelism and data parallelism [21–23].

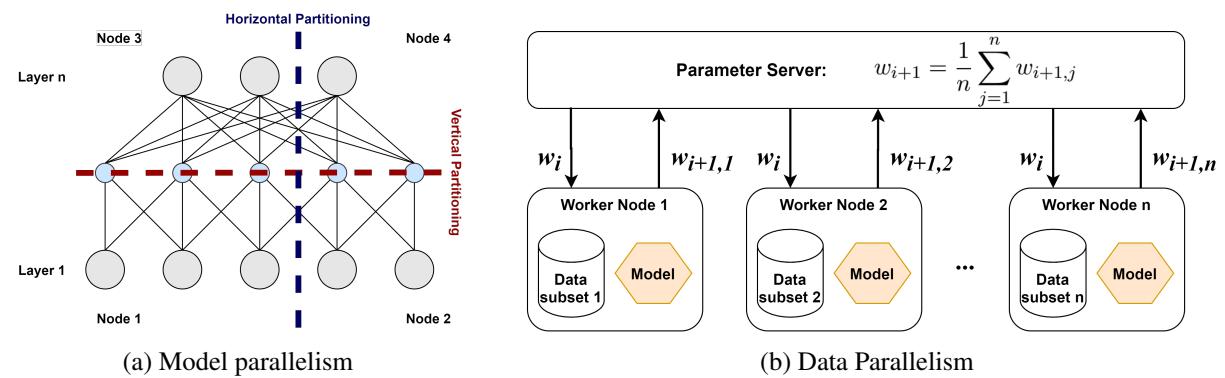


Fig. 2. Two distributed DNN training methods

In model parallelism, as in Fig. 2a, the model is divided into different parts that can run concurrently on different machines, and each part will run on the same dataset. The worker nodes only need to

synchronize the shared weights, usually once per propagation step forward or backward. Since this method is quite complicated, we should only use it when the large model does not fit in a single machine. Because of the complexity of this method and the limited scope of the paper, we only focus on the data parallelism method.

Data parallelism, as in Fig. 2b, is the most frequent method, as well as the simplest to build and use in most situations. In this approach, the training dataset is divided into partitions and distributed among worker nodes. The model is created at the master node, then sent to the worker nodes, and independently trained using previously distributed data partitions. After training the model for a controlled period, the model weights are sent to the master node by the worker nodes and the averaged values are used to update the weights of the model on the master node.

Data parallelism includes two main types: 1) Synchronous training (Fig. 3a): all worker nodes synchronize with each other during training; 2) Asynchronous training (Fig. 3b): each worker node trains at its own pace.

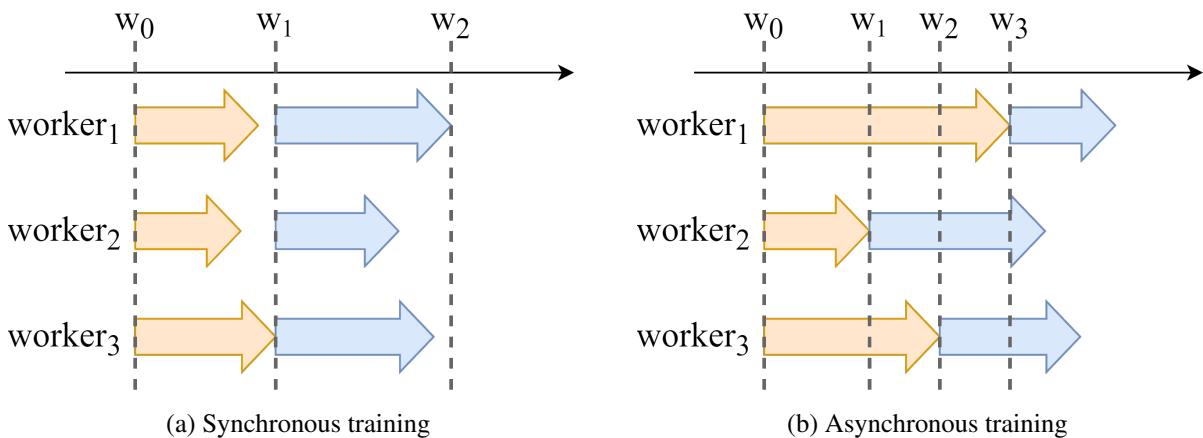


Fig. 3. Synchronous and asynchronous distributed DNN training

4. Methodology

4.1. Implementation of a distributed DNN training system on Apache Spark

Apache Spark is a specialized distributed processing framework for processing and analyzing big data very efficiently. Apache Spark has become the most commonly used big data processing framework today. However, when deploying a distributed DNN training application on Apache Spark, we encountered difficulties that adversely affected the entire system. Algorithm 1 shows the synchronous DNN training on Apache Spark.

Algorithm 1. Synchronous DNN training on K worker nodes on Apache Spark

```

1 function master_train () do
2   // Load the dataset
3   ( $x_{train}$ ,  $y_{train}$ ), ( $x_{test}$ ,  $y_{test}$ )  $\leftarrow$  load_dataset()
4   // Create the model
5   model  $\leftarrow$  create_model()

```

In the above implementation, there are three major problems, which adversely affect the performance of the entire system: 1) *In each iteration, shuffling the data, partitioning the data, and sending new partitions to the worker nodes is time-consuming*; 2) *Programming thinking is dependent on the structure map/reduce, programmers are not free to develop ideas*; 3) *The function worker_train() is sent from*

the master node to the worker nodes multiple times, while the contents of this function do not change, increasing the traffic on the network unnecessarily. Through deploying distributed DNN training application on Apache Spark, we encountered the following difficulties:

- Apache Spark's *map/reduce* structure is very effective in big data processing, but it is not suitable for the distributed DNN training because the programming is confined to the *map/reduce* structure, which binds programming thinking, resulting in the less flexible and ineffective implementation of DNN training applications.
- Because it depends on the *map/reduce* structure, the function *worker_train(data_iterator)* accepts only one parameter, *data_iterator* (which is a data partition). The programmer cannot pass other information to the *worker_train()* function as needed. Using an alternative is too troublesome. This makes it difficult to deploy applications.
- The mechanism for sharing data between a master node and worker nodes is very limited through *broadcast and accumulated variables*. While broadcast variables are read-only variables, and accumulated variables are addition-only variables.
- Users cannot actively add/remove variables or methods to worker nodes.
- Apache Spark does not support asynchronous communication between a master node and worker nodes. Therefore, to implement an asynchronous DNN training application, application developers must either manually add asynchronous communication to Apache Spark or use a framework that supports asynchronous communication.

4.2. Reasons for developing DDLF

To avoid the above difficulties when developing a distributed DNN training application on Apache Spark, we consider 2 solutions: 1) *Using an existing framework*; 2) *Building a new framework*. Currently, many frameworks support distributed DNN training such as Distributed TensorFlow, Elephas, Horovod, TensorFlowOnSpark, BigDL, PyTorch, Ray [2, 21]. Using an existing framework helps to develop applications quickly, but the downside is to accept its internal shortcomings, and difficult to improve, leading to less flexibility. We choose to build a new framework named DDLF because of the following reasons:

- Mastering the technology: The framework is developed by us, so we understand the techniques applied, understand its inner workings, and master the key techniques.
- Deepening the research content: When developing the framework, we have to conduct a lot of research, and surveys to deepen the content under research.
- Flexibility: We are not tied to any programming structure, such as *map/reduce*, which frees up programming thinking and helps to develop ideas freely and comfortably.
- Scalability: With the support of the techniques used in our framework, we can improve and extend our framework more easily.
- Overcoming the limitations encountered: Through our experience using existing frameworks, we can provide solutions to problems encountered.

4.3. The architecture of DDLF

The architecture of DDLF is a computer cluster consisting of a master node and many worker nodes as in Fig. 4. The master node will control the worker nodes to handle tasks in a distributed fashion. The *IWorker* interface specifies the functionality of the worker nodes. The worker nodes are implemented

1 using the *Worker* class. The master node consists of three main classes: 1) *ProxyWorker*: represents
 2 a worker node; 2) *Cluster*: represents the cluster; 3) *App*: represents a distributed application. The
 3 *Request* class represents a Remote Procedure Call (RPC) request which is sent from the master node to
 4 the worker nodes.

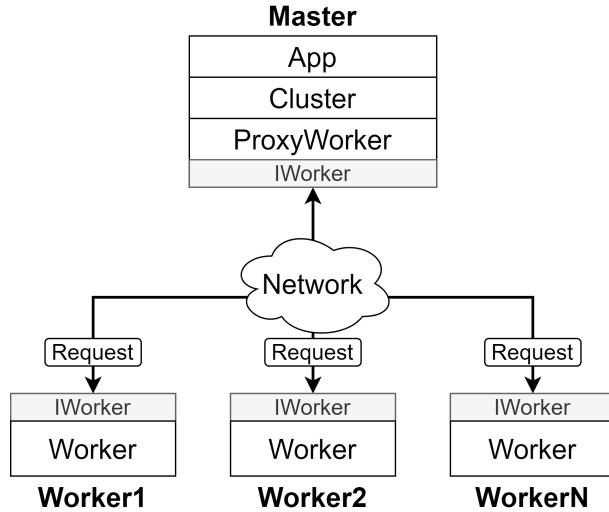


Fig. 4. The architecture of DDLF

4.4. Deploying a distributed DNN training application on DDLF

To deploy a distributed DNN training application on DDLF, we perform the steps as in Algorithm 2.

Algorithm 2. Deploying a distributed DNN training application on DDLF

```

1 function distributed_DNN_training_app() do
2     Create an object of the Cluster class
3     Create a connection between the master node and the cluster
4     Load a dataset
5     Create a model
6     Train the model
7     Close the connection between the master node and the cluster
8     Evaluate the model
9     Save the model
10 end function

```

For the convenience of deploying a distributed application, we have created the *IApp* interface to specify the required functions. When developing an application, the application developer only needs to create the *App* class (you can name it differently) that implements the *IApp* interface and override some of the necessary methods. Example 1 demonstrates a synchronous distributed DNN training application on DDLF.

 Example 1. A synchronous distributed DNN training application on DDLF

```

1  async function main() do
2      app ← App()
3      await app.connect()
4      await app.load_dataset()
5      await app.create_model()
6      await app.train_sync(master_epochs=100, worker_epochs=10, batch_size=32)
7      await app.evaluate_model()
8      await app.save_model(path)
9      await app.close()
10 end function

```

In the *App* class, the *load_dataset()* and *create_model()* methods are required to be implemented according to specific requirements. The methods *train()*, *train_sync()*, and *train_async()*, if the default implementation is accepted, we don't need to override them; otherwise, we can override them appropriately. Example 2 details the implementation of the *load_dataset()* method to load the MNIST dataset.

 Example 2. The *load_dataset()* method of the *App* class

```

20 async function load_dataset() do
21     // Load the MNIST dataset
22     (x_train, y_train), (x_test, y_test) ← load_mnist_dataset()
23     // Make sure images have shape (28, 28, 1)
24     (x_train, x_test) ← reshape(x_train, x_test)
25     // Normalize pixel values from [0, 255] to [-0.5, 0.5] to make it easier to work with
26     x_train, x_test ← (x_train / 255.0) - 0.5, (x_test / 255.0) - 0.5
27 end function

```

Example 3 details the implementation of the *create_model()* method to create a CNN model for classification on the MNIST dataset.

 Example 3. The *create_model()* method of the *App* class

```

34 async function create_model() do
35     // Set the input shape of MNIST
36     input_shape ← (28, 28, 1)
37     // Create the model
38     model ← models.Sequential()
39     model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
40     model.add(MaxPooling2D(pool_size=(2, 2)))
41     model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
42     model.add(Flatten())
43     model.add(Dense(64, activation='relu'))
44     model.add(Dense(10, activation='softmax'))
45 end function

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46

4.5. Features of DDLF

DDLF has the following advantages:

- (1) Simplicity: The design intent was to keep DDLF as simple as possible. Simple to install and use makes it easy to approach and quickly deploy the application, saving time and effort, and eventually, attaining economic efficiency. To install DDLF on the cluster just run a simple script. To deploy a distributed DNN training application, just extend the *IApp* interface to reuse existing utility methods, without complicated programming.
- (2) Efficient multitasking: DDLF uses the *asyncio* package of *Python 3.7+* to make multitasking more efficient than *multi-threading*, resulting in better processing efficiency.
- (3) Multi-platform: DDLF can be installed on different operating systems, as long as the operating system supports Python 3.7+.
- (4) Flexibility: DDLF allows the master node to control worker nodes without any restrictions. This facilitates free programming thinking, programming is not dependent or constrained by any programming structure.
- (5) Scalability: DDLF allows the master node to dynamically add variables and methods to worker nodes to serve diverse practical needs. Additional variables and methods persist on worker nodes until performing a request to delete or shut down the cluster. This allows the reusing of variables and methods, minimizing the need to transfer data or programs from the master node to the worker nodes, resulting in shorter processing times. This feature also helps programmers to dynamically and easily organize data and programs at worker nodes, so that distributed programming will be more natural, unrestricted, and increase performance. In addition, DDLF also allows adding worker nodes easily by making small changes in the configuration file.
- (6) Efficiency: Experimentally, DDLF proves to be effective because it helps to quickly train DNN and create a quality model.

DDLF also has the following disadvantages:

- (1) Multi-language not supported: Currently, DDLF only supports Python language.
- (2) The utility functions are not yet rich: The provision of many utility functions helps to quickly develop the application. However, in this release, there are a few utility functions, and we need to add more.
- (3) Optimizing communication costs: During the training of a distributed DNN, the communication between the master node and the worker nodes occurs frequently and often requires the transmission of large amounts of data such as gradients, and data. Therefore, a good framework must have measures to optimize communication costs. However, in this release, DDLF has not implemented communication cost optimization measures.
- (4) Training methods are not diversified: Currently, DDLF only supports training methods such as synchronous and asynchronous data parallelization.

4.6. Our solutions to overcome the obstacles in DDLF

As mentioned above, when implementing distributed DNN training systems on Apache Spark, we encountered some obstacles. Here are our solutions to overcome them.

4.6.1. Dynamic organization of data and functions at worker nodes

On DDLF, users can add/remove variables or methods to worker nodes without any constraints. This makes it easy for users to develop and deploy distributed applications. Meanwhile, Apache Spark allows to do this in a limited way via the *map/reduce structure* and *shared variables*.

A. The map/reduce structure

To solve problems when using the map/reduce structure, we allow the master node to send code freely to the worker nodes for execution. Code content can be a code segment or a method. In addition, we also allow the master node to dynamically add methods to worker nodes. These methods will persist on worker nodes until the master node requests to delete them or shut down the cluster. In the future, we will allow these additional methods to persist on worker nodes even after shutting down the cluster. This solution offers the following benefits:

- The code, which is sent to worker nodes, does not depend on any programming structure (like *map/reduce*). Methods can have any number of parameters and arbitrary parameter data types. This frees up programming thinking and facilitates the development of ideas.
- For methods that are used repeatedly, we don't need to send them to the worker nodes before each use as in the *map/reduce* structure but just send them to the worker nodes once, then use them again and again. This reduces the amount of traffic on the network.

B. Shared variables

In Apache Spark, the mechanism for sharing data between the master node and worker nodes is very limited via *broadcast* and *accumulated variables*. Broadcast variables are read-only variables, and accumulated variables are add-only variables. Furthermore, these variables have the same content across all worker nodes and they will be lost at the end of a Spark session.

To make the method of sharing data more convenient, on DDLF, we allow the master node to dynamically add variables to the worker nodes and manipulate them freely. The contents of these variables can be the same or different across worker nodes. Like additional methods, these variables will persist across worker nodes until the master node requests to delete them or shut down the cluster. In addition, the master node can also request worker nodes to access the local or shared file system to load or store data at the worker nodes. This solution has the following benefits:

- Convenient in organizing data at worker nodes, facilitating the development of ideas and implementation of algorithms.
- Since dynamically added variables can persist until shutting down the cluster, the master node only needs to send data to worker nodes once and then reuse it many times, even across different applications. This reduces communication costs between the master node and the worker nodes.

4.6.2. Processing of training data

To improve problems related to data shuffling, data partitioning, and sending data to worker nodes; we propose the following solution:

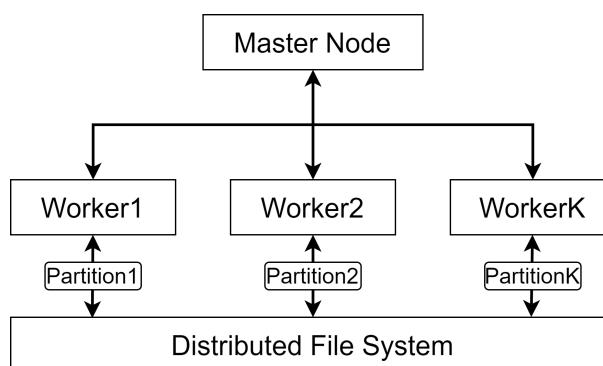
- Worker nodes will preload all training data if it is not already in the cache.
- At each iteration, the master node simply shuffles the index of the training dataset, partitions the index set, and sends the index partitions to the worker nodes.

- 1 Worker nodes will take their training data partition based on the previously loaded training dataset
2 and the received index partition.

3 This solution has the following advantages:
4

- 5 It minimizes the size of the data sent from the master node to the worker nodes because the size of
6 an index, which is an integer, is very small compared to the size of a training data sample, especially
7 image data. This results in a maximum reduction in data sending time, which greatly improves the
8 performance of the distributed DNN training system.
9
- 10 The datasets can be cached in the worker nodes' memory or local file system for reuse in different
11 DNN training applications.
12
- 13 It avoids creating a bottleneck at the master node because the master node does not have to distribute
large volumes of training data to all worker nodes.

14 The above solution well solves the case that the entire training dataset can be stored in one worker
15 node. However, when the size of the training dataset is too large, it exceeds the storage and processing
16 capacity of a worker node; we need to improve the above solution like Fig. 5. First, we store the training
17 dataset in a distributed file system accessible to all worker nodes (such as HDFS, S3), and partition it
18 into K partitions, where K is the number of worker nodes. A distributed file system can store very large
19 files [18, 19]. The whole dataset cannot fit in the memory of one worker node, but one partition can.
20 Then, each worker node will download its data partition to perform DNN training if there is no data
21 in the cache. Once loaded, these data partitions can be cached in the worker nodes' memory or local
22 file system for reuse. This solution is useful when a dataset is used for many different DNN training
23 applications, or when a DNN needs to be trained many times because you don't have to reload large
24 volumes of data. When needing to shuffle data, we will control the worker nodes to load a different
25 partition than the previous one, even re-partition the data. In case a partition also doesn't fit in the
26 memory of one worker node, we will expand the cluster by adding worker nodes.
27



39 Fig. 5. Loading datasets into DDLF
40

41 The above solutions are easily implemented on DDLF because of its flexibility (such as allowing
42 dynamically adding variables and methods to worker nodes, allowing access to the local file system
43 of worker nodes). However, on Apache Spark, it is very difficult for us to implement these solutions
44 because Apache Spark does not allow the master node to access the worker nodes' memory and local
45 file system flexibly.
46

1 **4.6.3. Asynchronous communication** 1

2 Apache Spark does not support asynchronous communication between the master node and the worker
 3 nodes. Therefore, to implement an asynchronous DNN training application, developers must either man-
 4 ually add asynchronous communication to Apache Spark or use a framework that supports asynchron-
 5 ous communication.
 6

7 On DDLF, the master node is free to control worker nodes' operations without any restrictions. So,
 8 when needed, the master node can very natively communicate with worker nodes synchronously or
 9 asynchronously.
 10

11 **5. Experiments**

12 **5.1. Datasets**

13 In this research, we use two datasets MNIST and CIFAR-10. They are well-known datasets, often
 14 used to train image processing, computer vision, and machine learning systems. The MNIST dataset
 15 (<http://yann.lecun.com/exdb/mnist/>) consists of 70,000 black and white images, belonging to 10 classes,
 16 representing 28x28 handwritten digits. In the MNIST dataset, the training dataset has 60,000 images and
 17 the test dataset has 10,000 images. The CIFAR-10 dataset (<https://www.cs.toronto.edu/~kriz/cifar.html>)
 18 consists of 60,000 32x32 color images, belonging to 10 classes, with 6,000 images for each class. The
 19 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. In
 20 the CIFAR-10 dataset, the training dataset has 50,000 images and the test dataset has 10,000 images.
 21

22 **5.2. Experimental system configuration**

23 We have built a cluster of nine computers. In which, one computer is the master node and eight
 24 computers are the worker nodes. The detailed configuration of the cluster is listed in Table 1.
 25

26 Table 1. Computer cluster configuration

#	Properties	Master Node	Worker Nodes
1.	Processor	Intel(R) Core™ i5-6500 CPU @ 3.20GHz	Intel(R) Core™ i5-6500 CPU @ 3.20GHz
2.	Cores	8	4
3.	Thread(s) per core	1	1
4.	RAM	16.0 GB	8.0 GB

42 We conducted experiments in two environments: a Spark cluster and a DDLF cluster. The Spark cluster
 43 is a computer cluster where each computer is installed with the Apache Spark distributed processing
 44 framework. Similarly, the DDLF Cluster is a computer cluster where each computer is installed with the
 45 DDLF distributed DNN training framework.
 46

5.3. Performance comparison between Apache Spark and DDLF

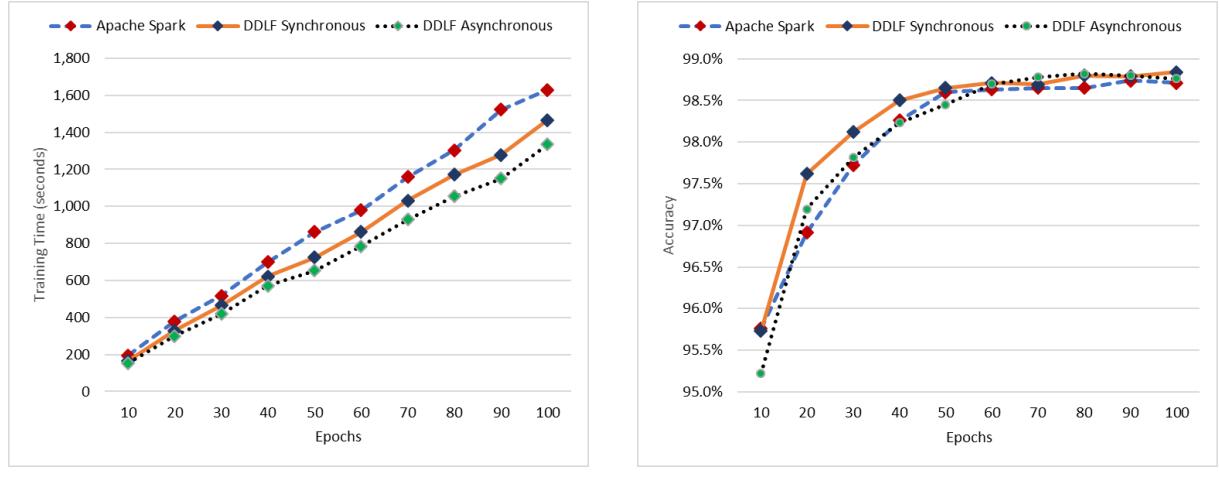
To compare the efficiency between Apache Spark and DDLF in distributed DNN training, we conducted three kinds of experiments on two datasets MNIST and CIFAR-10: 1) *Using Keras in combination with Apache Spark for synchronous training*; 2) *Use Keras in combination with DDLF for synchronous training*; 3) *Use Keras in combination with DDLF for asynchronous training*. Each type of experiment consists of ten different experiments based on the number of epochs varying from 10 to 100, with a step of 10. The CNN model is used for training as shown in Example 3. We did not test the case of using Keras in combination with Apache Spark for asynchronous training because Apache Spark does not support asynchronous communication between the master node and the worker nodes as mentioned in Section 4.1.

Table 2 shows the details of the experimental results on MNIST. Fig. 6 shows charts illustrating the experimental results. The chart in Fig. 6a shows that DDLF is more efficient than Apache Spark because of its faster training time. On the other hand, since asynchronous training takes full advantage of the worker nodes, it is more efficient than synchronous training. The chart in Fig. 6b shows that the accuracy of the model in all three cases is not significantly different.

Table 2. Comparison of the CNN model training time and accuracy in Apache Spark cluster and DDLF cluster on MNIST

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
10	195.57	95.76%	166.23	95.73%	152.93	95.22%
20	378.69	96.91%	329.46	97.62%	299.81	97.20%
30	517.14	97.72%	465.43	98.12%	418.88	97.81%
40	698.14	98.26%	621.34	98.50%	571.64	98.23%
50	863.14	98.60%	725.04	98.65%	652.53	98.45%
60	979.79	98.63%	862.22	98.71%	784.62	98.70%
70	1,158.79	98.65%	1,031.33	98.69%	928.19	98.78%
80	1,302.70	98.65%	1,172.43	98.80%	1,055.19	98.82%
90	1,523.03	98.74%	1,279.34	98.79%	1,151.41	98.80%
100	1,630.16	98.71%	1,467.14	98.84%	1,335.10	98.76%

Table 3 shows the details of the experimental results on CIFAR-10. Fig. 7 shows charts illustrating the experimental results. Similar to MNIST, experimental results on CIFAR-10 also show that DDLF is more efficient than Apache Spark. The accuracy of the model in all three cases is also almost the same. On the other hand, because CIFAR-10 (containing color images) is more complex than MNIST (containing only black and white images), the training time on CIFAR-10 is much longer than on MNIST.



(a) Comparison of training time

(b) Comparison of model accuracy

Fig. 6. Comparison of the CNN model training time and accuracy in Apache Spark cluster and DDLF cluster on MNIST

Table 3. Comparison of the CNN model training time and accuracy in Apache Spark cluster and DDLF cluster on CIFAR-10

Epochs	Apache Spark		DDLF Synchronous		DDLF Asynchronous	
	Time	Accuracy	Time	Accuracy	Time	Accuracy
10	309.57	56.31%	275.52	57.31%	250.72	55.68%
20	644.34	64.42%	579.90	66.81%	533.51	65.48%
30	810.83	68.91%	689.21	70.26%	634.07	69.56%
40	1,303.50	72.01%	1,094.94	73.23%	985.45	72.49%
50	1,587.21	72.21%	1,428.49	73.86%	1,299.92	73.56%
60	1,876.13	74.09%	1,575.95	75.00%	1,434.12	74.25%
70	2,064.50	73.66%	1,775.47	74.82%	1,633.43	72.57%
80	2,179.49	74.96%	1,874.36	75.37%	1,705.67	74.38%
90	2,486.07	74.62%	2,088.30	74.95%	1,879.47	73.45%
100	2,785.08	74.81%	2,506.57	75.59%	2,255.91	75.34%

By observing the experimental results on both data sets, we can summarize as follows: 1) The performance of DDLF compared to Spark becomes more apparent as the number of epochs is increased or the training dataset is larger. 2) The asynchronous training method always has better performance than the synchronous training method. 3) The accuracy of the model in all three ways of training is almost the same. That is the result of the solutions that we have applied to DDLF. Table 4 summarizes the experi-

mental results on two datasets MNIST and CIFAR-10. On average, synchronous DDLF is 12.80% faster than Apache Spark, asynchronous DDLF is 20.78% faster than Apache Spark, and asynchronous DDLF is 9.15% faster than synchronous DDLF.

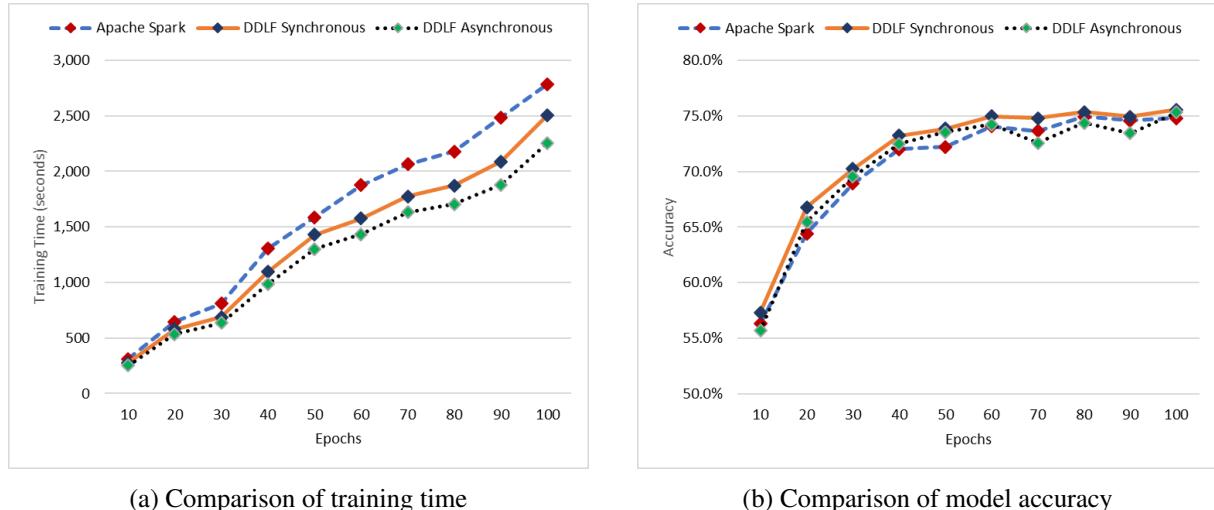


Fig. 7. Comparison of the CNN model training time and accuracy in Apache Spark cluster and DDLF cluster on CIFAR-10

Table 4. Summary of the comparison of training time on two datasets

Descriptions	Average per Dataset		Average
	MNIST	CIFAR-10	
DDLF Synchronous is faster than Apache Spark	12.40%	13.20%	12.80%
DDLF Asynchronous is faster than Apache Spark	20.55%	21.01%	20.78%
DDLF Asynchronous is faster than DDLF Synchronous	9.30%	9.00%	9.15%

6. Conclusions and future works

In the big data era, small training datasets are increasingly rare. Training DNN in a distributed environment is the goal to aim because of its practical benefits. Apache Spark is considered one of the most prominent big data processing frameworks today. Therefore, many DNN training frameworks have combined with Apache Spark to take advantage of its distributed computing feature. However, the design purpose of Apache Spark is to handle big data, not to train DNN. The forced use of Apache Spark to distributed DNN training faces several challenges, resulting in a reduction in overall system performance. So, we developed the DDLF distributed processing framework, which is completely independent of Apache Spark, overcoming the obstacles we encountered when using Apache Spark to train DNN. Although the original goal of DDLF design was to specialize in training distributed DNN, DDLF can

also be used to deploy any type of distributed application. Advantages of the DDLF framework: 1) Simple, easy to deploy distributed application; 2) Make programming natural, and comfortable; 3) Flexible and extensible; 4) High performance. Besides, DDLF also has the following disadvantages: 1) Currently, only Python programming language is supported and requires version from 3.7 onwards due to the inside technique using package `asyncio`; 2) The utility functions are not many because they are newly developed; 3) The communication cost between the master node and the worker nodes is not optimized.

We will continue to research and improve DDLF as follows: 1) Integrate some important components written in C/C++ to improve performance; 2) Add more utility functions to support better-distributed training; 3) Research to optimize communication costs between the master node and the worker nodes.

In addition, we will study using DDLF to train and fine-tune our BERT model and implement a DNN training system using model parallelism.

Conflict of interest

We have no conflict of interest for this paper.

Acknowledgment

This research is funded by Vietnam National University Ho Chi Minh City (VNU-HCMC) under grant number DS2020-26-01.

References

- [1] C. Sun, A. Shrivastava, S. Singh and A. Gupta, Revisiting Unreasonable Effectiveness of Data in Deep Learning Era, 2017.
- [2] J. Verbraecken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen and J.S. Rellermeyer, A Survey on Distributed Machine Learning, *ACM Comput. Surv.* **53**(2) (2020). doi:10.1145/3377454.
- [3] V. Campos, F. Sastre, M. Yagües, M. Bellver, X. Giró-i-Nieto and J. Torres, Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster, *Procedia Computer Science* **108** (2017), 315–324, International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. doi:<https://doi.org/10.1016/j.procs.2017.05.074>. <https://www.sciencedirect.com/science/article/pii/S1877050917306129>.
- [4] C.-C. Chen, C.-L. Yang and H.-Y. Cheng, Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform, 2019.
- [5] P. Do, T.P. Hong and H.D. To, DMTree: A Novel Indexing Method for Finding Similarities in Large Vector Sets, *International Journal of Advanced Computer Science and Applications* **11**(4) (2020). doi:10.14569/IJACSA.2020.0110483.
- [6] T. Phan and P. Do, Building a Vietnamese question answering system based on knowledge graph and distributed CNN, *Neural Computing and Applications* (2021). <https://doi.org/10.1007/s00521-021-06126-z>.
- [7] P. Do, T. Phan, H. Le and B.B. Gupta, Building a knowledge graph by using cross-lingual transfer method and distributed MinIE algorithm on apache spark, *Neural Computing and Applications* (2020). <https://doi.org/10.1007/s00521-020-05495-1>.
- [8] M. Li, D.G. Andersen, J.W. Park, A.J. Smola, A. Ahmed, V. Josifovski, J. Long, E.J. Shekita and B.-Y. Su, Scaling Distributed Machine Learning with the Parameter Server, in: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, USENIX Association, USA, 2014, pp. 583–598-. ISBN 9781931971164.
- [9] M. Li, D.G. Andersen, A.J. Smola and K. Yu, Communication Efficient Distributed Machine Learning with the Parameter Server, in: *Advances in Neural Information Processing Systems*, Vol. 27, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence and K.Q. Weinberger, eds, Curran Associates, Inc., 2014. <https://proceedings.neurips.cc/paper/2014/file/1ff1de774005f8da13f42943881c655f-Paper.pdf>.
- [10] 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, 2016.
- [11] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger and P. Gibbons, PipeDream: Fast and Efficient Pipeline Parallel DNN Training, 2018.
- [12] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan and I. Stoica, Ray: A Distributed Framework for Emerging AI Applications, 2018.

- 1 [13] A. Wang, X. Jia, L. Jiang, J. Zhang, Y. Li and W. Lin, Whale: A Unified Distributed Training Framework, *CoRR abs/2011.09208* (2020). <https://arxiv.org/abs/2011.09208>. 1
- 2 [14] R. Rajak, Cluster Computing: Emerging Technologies, Benefits, Architecture, Tools and Applications, *International Journal of Advanced Science and Technology* **29**(04) (2020), 4008 -. <http://sersc.org/journals/index.php/IJAST/article/view/24569>. 2
- 3 [15] F. Es-Sabery and A. Hair, Big Data Solutions Proposed for Cluster Computing Systems Challenges: A Survey, in: *Proceedings of the 3rd International Conference on Networking, Information Systems & Security*, NISS2020, Association for 3
- 4 Computing Machinery, New York, NY, USA, 2020. ISBN 9781450376341. doi:10.1145/3386723.3387826. 4
- 5 [16] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine and S. Khan, A Survey of Distributed Data Stream Processing 5
- 6 Frameworks, *IEEE Access* **7** (2019), 154300–154316. 6
- 7 [17] E. Shaikh, I. Mohiuddin, Y. Alufaisan and I. Nahvi, Apache Spark: A Big Data Processing Engine, in: *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, 2019, pp. 1–6. 7
- 8 doi:10.1109/MENACOMM46666.2019.8988541. 8
- 9 [18] K. Shvachko, H. Kuang, S. Radia and R. Chansler, The Hadoop Distributed File System, in: *2010 IEEE 26th Symposium 9*
- 10 on Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10. doi:10.1109/MSST.2010.5496972. 10
- 11 [19] V. Persico, A. Montieri and A. Pescapè, On the Network Performance of Amazon S3 Cloud-Storage Service, 2016, 11
- 12 pp. 113–118. doi:10.1109/CloudNet.2016.16. 12
- 13 [20] O.G. Yalçın, Top 5 Deep Learning Frameworks to Watch in 2021 and Why TensorFlow, Medium, accessed 6/8/2021. 13
- 14 [21] M. Langer, Z. He, W. Rahayu and Y. Xue, Distributed Training of Deep Learning Models: A Taxonomic Perspective, 14
- 15 *IEEE Transactions on Parallel and Distributed Systems* **31**(12) (2020), 2802–2818. doi:10.1109/TPDS.2020.3003307. 15
- 16 [22] A. Sharma, Guided parallelized stochastic gradient descent for delay compensation, *CoRR abs/2101.07259* (2021). <https://arxiv.org/abs/2101.07259>. 16
- 17 [23] K.S. Chahal, M.S. Grover, K. Dey and R.R. Shah, A Hitchhiker’s Guide On Distributed Training Of Deep Neural Networks, *Journal of Parallel and Distributed Computing* **137** (2020), 65–76. doi:<https://doi.org/10.1016/j.jpdc.2019.10.004>. 17
- 18 <https://www.sciencedirect.com/science/article/pii/S0743731518308712>. 18
- 19 20
- 21 21
- 22 22
- 23 23
- 24 24
- 25 25
- 26 26
- 27 27
- 28 28
- 29 29
- 30 30
- 31 31
- 32 32
- 33 33
- 34 34
- 35 35
- 36 36
- 37 37
- 38 38
- 39 39
- 40 40
- 41 41
- 42 42
- 43 43
- 44 44
- 45 45
- 46 46