

A DISSERTATION  
SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY  
IN COMPUTER SCIENCE AND ENGINEERING

**Advanced Distributed System-Algorithm  
Co-Design for Graph Learning**



by

Fahao Chen

*June 2024*

© Copyright by Fahao Chen, June 2024

All Rights Reserved.

The thesis titled

*Advanced Distributed System-Algorithm Co-Design for Graph Learning*

by

Fahao Chen

is reviewed and approved by:

---

**Chief referee**

*Senior Associate Professor*

Date

*Peng Li*

---

*Professor*

Date

*Anh Tuan Pham*

---

*Professor*

Date

*Rentaro Toshioka*

---

*Senior Associate Professor*

Date

*Yoichi Tomioka*

---

THE UNIVERSITY OF AIZU

*June 2024*

The thesis titled

*Advanced Distributed System-Algorithm Co-Design for Graph Learning*

by

Fahao Chen

is reviewed and approved by:

**Chief referee**

*Senior Associate Professor*

Date

2024/8/12

Peng Li

*Peng Li*



*Professor*

Date

2024/8/12

Anh Tuan Pham

*Pham T. Anh*



*Professor*

Date

2024/8/13

Rentaro Yoshioka

*R. Yoshioka*



*Senior Associate Professor*

Date

2024/8/13

Yoichi Tomioka

*Yoichi Tomioka*



THE UNIVERSITY OF AIZU

June 2024



# Contents

<b>Chapter 1</b>	<b>Introduction</b>	<b>1</b>
<b>Chapter 2</b>	<b>Background</b>	<b>5</b>
2.1	Graphs and Graph Neural Networks . . . . .	5
2.2	Dynamic Graphs and Dynamic Graph Neural Networks . . . . .	6
2.3	Distributed Graph Learning . . . . .	8
2.4	Graph Partitioning . . . . .	9
2.5	Job Scheduling . . . . .	10
2.6	Federated Learning . . . . .	10
2.7	MoE Architecture . . . . .	11
2.8	Offloading-based model serving. . . . .	11
2.9	Tiny ML-based model serving. . . . .	12
<b>Chapter 3</b>	<b>Efficient Distributed Graph Learning on Heterogeneous GPUs</b>	<b>13</b>
3.1	Problem Statement . . . . .	13
3.2	Motivation . . . . .	16
3.2.1	GPU heterogeneity and inter-job parallelism. . . . .	16
3.2.2	GPU heterogeneity and intra-job parallelism. . . . .	17
3.2.3	Scale-fixed synchronization versus scale-adaptive synchronization	17
3.2.4	Task Switching Cost . . . . .	18
3.3	System Overview . . . . .	19
3.4	Fast Task Switching . . . . .	21
3.5	Task Scheduling Algorithm . . . . .	22
3.5.1	System Model . . . . .	22
3.5.2	Algorithm Design . . . . .	24
3.5.3	Theoretical Analysis . . . . .	26
3.6	Implementation . . . . .	28
3.7	Evaluation . . . . .	28
3.7.1	Experimental settings . . . . .	28
3.7.2	Results on Testbed . . . . .	30
3.7.3	Results of Simulations . . . . .	31
3.8	Summary . . . . .	32
<b>Chapter 4</b>	<b>Efficient Distributed Graph Learning on Dynamic Graphs</b>	<b>33</b>
4.1	Problem Statement . . . . .	33
4.2	Motivation . . . . .	36
4.2.1	Spatio-temporal non-uniformity of dynamic graphs . . . . .	36
4.2.2	Performance of dynamic graph partitioning methods on differ- ent datasets . . . . .	36

4.2.3	Performance of dynamic graph partitioning methods within a single dataset . . . . .	38
4.3	System Overview . . . . .	38
4.4	Partitioning by Graph Chunks . . . . .	40
4.4.1	Chunk Generation . . . . .	40
4.4.2	Chunk Assignment . . . . .	41
4.5	Run-time Optimization . . . . .	43
4.5.1	Chunk Fusion . . . . .	43
	Spatial Fusion . . . . .	43
	Temporal Fusion . . . . .	44
4.5.2	Adaptive Stale Embedding Aggregation . . . . .	44
4.6	Implementation . . . . .	46
4.7	Evaluation . . . . .	47
4.7.1	Experiment Setup . . . . .	47
4.7.2	Overall Performance Comparison . . . . .	48
4.7.3	Ablation Study . . . . .	49
	Impact of Partitioning by Graph Chunks (PGC) module . . . . .	49
	Impact of chunk fusion module . . . . .	50
	Impact of adaptive stale aggregation module . . . . .	51
4.7.4	Chunk workload prediction. . . . .	52
4.7.5	DGC Overhead . . . . .	53
4.7.6	Convergence Evaluation . . . . .	53
4.8	Summary . . . . .	54
<b>Chapter 5</b>	<b>Efficient Distributed Graph Learning with Privacy Preserving</b>	<b>55</b>
5.1	Problem Statement . . . . .	55
5.2	Motivation . . . . .	57
5.3	FedGraph Design . . . . .	58
5.3.1	Local Graph Convolutional Network (GCN) Training by Clients	59
	Graph Convolutional Network (GCN) construction . . . . .	61
	Graph Convolutional Network (GCN) training . . . . .	61
5.3.2	Global Parameter Update by the Server . . . . .	62
5.3.3	Security Analysis . . . . .	62
5.4	Intelligent Graph Sampling based on Deep Reinforcement Learning (DRL)	63
5.4.1	Deep Deterministic Policy Gradient (DDPG)-based problem for- mulation . . . . .	63
5.4.2	Sampling based on Deep Deterministic Policy Gradient (DDPG)	64
5.5	Evaluation . . . . .	66
5.5.1	Experimental settings . . . . .	66
5.5.2	Experimental results . . . . .	68
5.6	Summary . . . . .	71
<b>Chapter 6</b>	<b>Efficient Giant Model Inference on Edges using Graph Learning</b>	<b>75</b>
6.1	Problem Statement . . . . .	75
6.2	Motivation . . . . .	77
6.3	System Overview . . . . .	78
6.4	Problem Formulation . . . . .	79
6.5	Graph Learning-based Expert Feeding . . . . .	83

6.5.1	Graph Construction . . . . .	83
6.5.2	GESolver Overview . . . . .	84
6.5.3	GNN Based Feature Extractor . . . . .	85
6.5.4	MLP-based Solution Generator . . . . .	85
6.5.5	Algorithm Analysis . . . . .	86
6.6	Evaluation . . . . .	86
6.7	Summary . . . . .	91
<b>Chapter 7</b>	<b>Conclusion and Future Works</b>	<b>92</b>

# List of Figures

Figure 1.1 An overview of our three works on distributed graph learning. . . . .	2
Figure 2.1 Illustration of graph convolution operation. . . . .	6
Figure 2.2 DGNN execution flow. . . . .	7
Figure 2.3 DGNN layer structures. . . . .	7
Figure 2.4 Distributed Graph Neural Network (GNN) training. (a) An example of Graphs and Graph Neural Network (GNN)s, where NN indicates the neural network. (b) Illustration of distributed Graph Neural Network (GNN) training, where $W_i$ means the weights of $i$ -th Graph Neural Network (GNN) layer. . . . .	8
Figure 2.5 The illustration of the MoE architectures. . . . .	11
Figure 3.1 A toy example to show job scheduling results under different methods. (a) GPU-heterogeneity-oblivious scheduling result; (b) GPU-heterogeneity-aware scheduling result, but without exploiting intra-job parallelism; (c) A better scheduling result jointly considering GPU heterogeneity and intra-job parallelism. . . . .	15
Figure 3.2 Training speedup of different jobs on different GPUs. . . . .	16
Figure 3.3 The GPU utilization of training GraphSAGE model. . . . .	16
Figure 3.4 An example showing the benefit of relaxed scale-fixed synchronization scheme adopted by Hare. . . . .	17
Figure 3.5 Epoch time of ResNet152 under different GPU combinations. . . . .	18
Figure 3.6 GPU utilization of V100 and K80 when training ResNet152. . . . .	18
Figure 3.7 Ratio of switching time and training time under three settings. . . . .	19
Figure 3.8 V100 GPU utilization with and without task switching. . . . .	19
Figure 3.9 System Overview. . . . .	20
Figure 3.10 An example of speculative memory management. . . . .	22
Figure 3.11 Training two popular models on 8 V100 GPUs. . . . .	22
Figure 3.12 The results in testbed. . . . .	30
Figure 3.13 CDF of job completion time. . . . .	30
Figure 3.14 Performance under different number of GPUs. . . . .	30
Figure 3.15 Performance under different number of jobs. . . . .	30
Figure 3.16 Performance under different heterogeneity levels. . . . .	31
Figure 3.17 Performance under different fractions of jobs. . . . .	31
Figure 3.18 Performance under different bandwidth. . . . .	32
Figure 3.19 Performance under different batch sizes. . . . .	32
Figure 4.1 Dynamic graph neural network. . . . .	34

Figure 4.2	Different dynamic graph partitioning methods for distributed training. Different node shapes (such as rectangles, circles, triangles, and rhombuses) represent different vertices, while colors and numbers signify vertices belonging to different snapshots. Red dotted arrows indicate communication between GPUs. Within each GPU, vertices aggregated from other GPUs are represented by dotted lines. . . . .	35
Figure 4.3	CDF of number of spatial neighbors. . . . .	36
Figure 4.4	CDF of vertex sequence lengths. . . . .	36
Figure 4.5	Performance of dynamic graph partitioning methods on different datasets. . . . .	37
Figure 4.6	Performance of dynamic graph partitioning methods within a single dataset. . . . .	37
Figure 4.7	System Overview. DGC contains an offline Partitioning by Graph Chunks (PGC) module and an optimized run-time. The Partitioning by Graph Chunks (PGC) module partitions the dynamic graph into chunks and assigns them to GPUs. The run-time contains two new modules of chunk fusion and adaptive stale embedding aggregation. . . . .	39
Figure 4.8	The execution time of different chunks with the same number of vertices. . . . .	42
Figure 4.9	An illustration of spatial fusion. . . . .	43
Figure 4.10(a)	An illustration of temporal fusion. (b) We take a GRU layer as an example, which is adopted in the time encoder of T-GCN. The sequence is $(B1, B2, C1, C2)$ , given in Figure 4.10(a)(c). . . . .	44
Figure 4.11	CDF of L2 distances between embeddings. . . . .	45
Figure 4.12	Epoch time of different methods. . . . .	48
Figure 4.13	Synthetic dynamic graphs with spatial non-uniformity. . . . .	50
Figure 4.14	Synthetic dynamic graphs with temporal non-uniformity. . . . .	50
Figure 4.15	Chunk fusion performance. . . . .	51
Figure 4.16	Chunk workload prediction error and workload divergence. . . . .	52
Figure 4.17	Extra overhead introduced by DGC. The numbers above bars are the percentages of total training time. . . . .	53
Figure 4.18	Training loss and test accuracy under different methods for three models on the Epinion dataset. . . . .	53
Figure 5.1	An illustration of different sampling approaches. The sampled nodes are marked in color (dark, red, and blue). The dashed arrows denote edge connections in the original graph. The solid arrows denote the edges preserved by sampled nodes. . . . .	57
Figure 5.2	The FedGraph architecture. Each client $i$ maintains a local graph $G_i$ . During the training, nodes in the mini-batch (nodes in red) aggregate neighbors' embeddings to generate the next layer's embeddings, denoted by red arrows. When training completes, each client $i$ uploads its local model weights $W_i$ to the parameter server. Finally, the parameter server aggregates all local model weights to the updated global model $\bar{W}$ and sends it back to all clients. . . . .	58
Figure 5.3	System design. . . . .	59

Figure 5.4	Illustration of Deep Deterministic Policy Gradient (DDPG)-based sampling. . . . .	65
Figure 5.5	Cumulative discounted returns of FedGraph. . . . .	68
Figure 5.6	Accuracy convergence of different sampling schemes with 20 clients. Note that, FastGCN completes the training with less time as it samples fewer nodes for training. However, it has poor performances in all datasets. . . . .	69
Figure 5.7	Accuracy convergence of different sampling schemes with 50 clients. . . . .	70
Figure 5.8	The convergence time under different levels of graph heterogeneity. . . . .	71
Figure 5.9	Convergence of FedGraph and FedGraph_nonShare. FedGraph_nonShare completes the training with less time as it ignores lots of connections in the local training. However, it has a poor convergence. . . . .	72
Figure 5.10	Training accuracy and time under different Graph Convolutional Network (GCN) depths. . . . .	73
Figure 5.11	Accuracy convergence of different sampling schemes on non-iid data. . . . .	74
Figure 6.1	The illustration of the proposed inference system of MoE-based models on UAVs. Multiple UAVs download experts from edge servers via wireless networks. The downloaded experts are then used for various inference tasks on UAVs. . . . .	76
Figure 6.2	Frequencies of expert activation of two MoE-based ViT models on ImageNet dataset. Inference data will sparsely activate different experts. . . . .	78
Figure 6.3	An overview of the proposed MoE-based giant model serving system. . . . .	79
Figure 6.4	The graph to represent the expert feeding problem, where UAVs, servers, and experts are heterogeneous vertices. . . . .	83
Figure 6.5	The illustration of our proposed graph learning-based method, termed GESolver, which consists of a GNN-based feature extractor and an MLP-based solution generator. . . . .	84
Figure 6.6	The architecture of the GNN-based feature extractor. . . . .	85
Figure 6.7	The architecture of the MLP-based solution generator. . . . .	86
Figure 6.8	Overall performance under different number of UAVs. . . . .	88
Figure 6.9	Overall performance under different number of servers. . . . .	88
Figure 6.10	Overall performance under different number of experts. . . . .	89
Figure 6.11	Overall performance under different bandwidth. . . . .	89
Figure 6.12	Overall performance under different other settings. For example, $(C, S) = (256, 16)$ indicates a setting of a 256MB of capacity and each expert has a size of 16MB. . . . .	90
Figure 6.13	Training loss of GESolver. . . . .	90
Figure 6.14	The time costs of different algorithms under different problem settings. For example, $(100, 10, 24)$ indicates a setting of 100 UAVs, 10 servers, and 24 experts. . . . .	91

# List of Tables

Table 3.1	Notations . . . . .	23
Table 3.2	Deep Learning Jobs Used in Our Experiments. . . . .	29
Table 3.3	Average Task Switching Time of Different Jobs. . . . .	29
Table 4.1	Dynamic Graph Datasets. . . . .	36
Table 4.2	Test accuracy with different threshold $\theta$ . . . . .	46
Table 4.3	Impact of adaptive stale embedding aggregation. . . . .	51
Table 5.1	Graph Data Statistics . . . . .	67
Table 6.1	Notations . . . . .	82
Table 6.2	System Parameters . . . . .	87

# List of Abbreviations

<b>CNN</b>	Convolutional Neural Network
<b>DDPG</b>	Deep Deterministic Policy Gradient
<b>DGNN</b>	Dynamic Graph Neural Network
<b>DML</b>	Distributed Machine Learning
<b>DRL</b>	Deep Reinforcement Learning
<b>GCN</b>	Graph Convolutional Network
<b>GNN</b>	Graph Neural Network
<b>HE</b>	Homomorphic Encryption
<b>MPC</b>	Multi-party Computation
<b>PGC</b>	Partitioning by Graph Chunks
<b>PS</b>	Parameter Server
<b>PSS</b>	Partitioning by Spatial Snapshots
<b>PTS</b>	Partitioning by Temporal Sequences
<b>TEE</b>	Trusted Execution Environmet



# Acknowledgment

I am deeply grateful to all those who have supported and guided me throughout my doctoral studies.

First and foremost, I would like to express my heartfelt thanks to my supervisor, Prof. Peng Li. His patience, encouragement, and unwavering support have been invaluable throughout my research journey. I am also profoundly thankful to my dissertation committee—Prof. Anh T. Pham, Prof. Rentaro Yoshioka, and Prof. Yoichi Tomioka—whose insightful guidance and wisdom have been instrumental in shaping my research work.

I would also like to acknowledge the members of my lab and my friends at the university. Their constant support, valuable advice, and camaraderie have enriched both my research and my experience.

Finally, I am deeply indebted to my family. Their unconditional love, patience, and understanding have been my source of strength and motivation during my doctoral period. This thesis is dedicated to them.

# Abstract

This dissertation delves into the realm of graph learning, a pivotal area in the field of machine learning that leverages the vast yet complex graph data generated by edge devices such as smartphones and sensors. Graph data is typically composed of vertices (nodes) and edges, such as social networks, web graphs, and traffic networks. The processing of such data through Graph Neural Network (GNN) enables the extraction of valuable structural information, crucial for various applications, such as travel time predictions and protein prediction tasks. Despite GNNs' potential, their application to large-scale graphs is hindered by substantial computational and memory requirements, exceeding the capacities of conventional accelerators, e.g., GPUs.

This dissertation seeks to holistically enhance the efficiency of distributed graph learning through three novel works. Both of our first two works focus on the distributed graph training within the single data center while our third work moves to the setting of multiple data centers. Specifically, our first work addresses the challenges of training large graphs on heterogeneous GPUs. Here, the primary obstacle is the effective synchronization of models across GPUs that differ in capabilities. In addition, the training efficiency will also be influenced by the competition for limited GPU resources by multiple training jobs. The focus is on devising an efficient scheduling strategy that minimizes job completion times while optimizing resource utilization.

The second work targets the unique challenges posed by dynamic graphs, which evolve over time and typically involve larger graph sizes than static graphs considered in the first work. The main difficulty arises from the extensive edge connections that span spatial and temporal boundaries, escalating communication costs among distributed GPUs. To address this, we propose a novel graph partitioning method aimed at reducing inter-GPU traffic and enhancing the efficiency of distributed training.

Then, our third work expands the scope to the training of giant graphs distributed across multiple data centers, with a focus on privacy-preserving. Each data center holds a part of the giant graph, but privacy regulations such as GDPR and CCPA restrict the sharing of graph data among them. Training a GNN model solely on local graph data typically falls short in performance. Therefore, our final work concentrates on developing methods that keep privacy constraints while aiming to achieve high model performance across the distributed graph.

Lastly, we focus on graph learning to solve real-life problems. We focus on the giant model inference on resource-constrained devices. Our final work proposes a novel inference framework that decouple the giant model into multiple tiny experts. However, deciding the expert downloading and device-server association to enhance the inference performance is still challenging due to the high expert downloading delay and inherent iterations for expert downloading between edge devices.

To address the above challenges, we first introduce Hare, a system designed to optimize distributed graph learning on heterogeneous GPUs. Hare efficiently schedules

GPU resources across multiple training jobs to maximize resource utilization. It incorporates a scheduling algorithm based on a relaxed scale-fixed distributed training synchronization scheme and a fast task switching mechanism, both of which substantially decrease the completion time of training jobs. Next, we present DGC, a system specifically optimized for training on dynamic graphs. DGC innovates with a new partitioning method that partitions the dynamic graph into chunks. In addition, the training efficiency is further enhanced through two runtime optimization techniques. Then, we introduce FedGraph, a solution designed for training giant graphs across distributed data centers while preserving privacy. FedGraph employs a federated graph learning algorithm that allows multiple data centers to collaboratively train a global GNN model without sharing graph data. Additionally, it enhances the training process with a DRL-based intelligent sampling method. Finally, we propose GESolver, based on graph learning, which automatically decides which experts should be loaded from which servers during local inference on edge devices, to maximize the inference accuracy while minimizing expert loading costs. We conduct extensive experiments to evaluate our works, and the results demonstrate significant improvements over state-of-the-art methods.

# Chapter 1

## Introduction

Recently, the field of machine learning has made significant strides in exploiting the hidden values within data generated by edge devices, such as smartphones and sensors. In addition to the traditional data forms like images and sentences, a unique and important type of learning data, known as *graphs*, has gained increasing attention. Graph data, e.g., social networks, web graphs, and traffic networks, is generally composed of vertices (or nodes) and edges. For instance, within an e-commerce platform like Amazon, users and products form the vertices, while the interactions between them, such as browsing or purchasing, are represented by the edges. Currently, the most advanced algorithm for processing graph data is the GNN, which employs neural networks to extract the hidden structural information embedded in graphs. The application of graph learning spans numerous fields. For instance, Google Maps leverages it for enhanced travel time predictions, Decathlon Canada for personalized sports product recommendations, and Amazon has developed tools like SageMaker and DocumentDB to facilitate graph learning in protein prediction tasks.

Recently, large-scale GNNs have become more and more important due to the abundance of massive graph data. Common examples include the Microsoft Academic Graph (MAG), which consists of 111 million vertices and 3.23 billion edges, and Pinterest’s user-item graph, with over 2 billion vertices and 17 billion edges, totaling a data size of 18TB. However, most existing GNN models are evaluated only on smaller datasets, making it challenging or inefficient to process such large graphs due to the complexity and computational demands of GNN models. To address this, one approach focuses on designing more efficient GNN models through techniques like simplification [1–3], quantization [4–9], sampling [10–12], and distillation [13–15]. Another approach leverages distributed computing for GNN training, a.k.a., distributed graph learning [16–20]. This method is particularly beneficial for handling large graphs where the limited memory and computing resources of a single device, such as a GPU, become bottlenecks. Therefore, utilizing distributed computing can substantially improve training efficiency with resources like multi-GPUs or CPU clusters.

In this dissertation, we aim to improve the efficiency of distributed graph learning comprehensively. Figure 1.1 includes three main works in this dissertation. From the training scale perspective, we first focus on the single data center, which includes multiple training accelerators, i.e., GPUs. The data center owns a large graph and conducts distributed graph learning with multiple GPUs. In this process, we mainly consider the efficiency of the distributed graph learning job while aiming to optimize resource utilization. Then, we move to multiple data centers scale, where different data centers

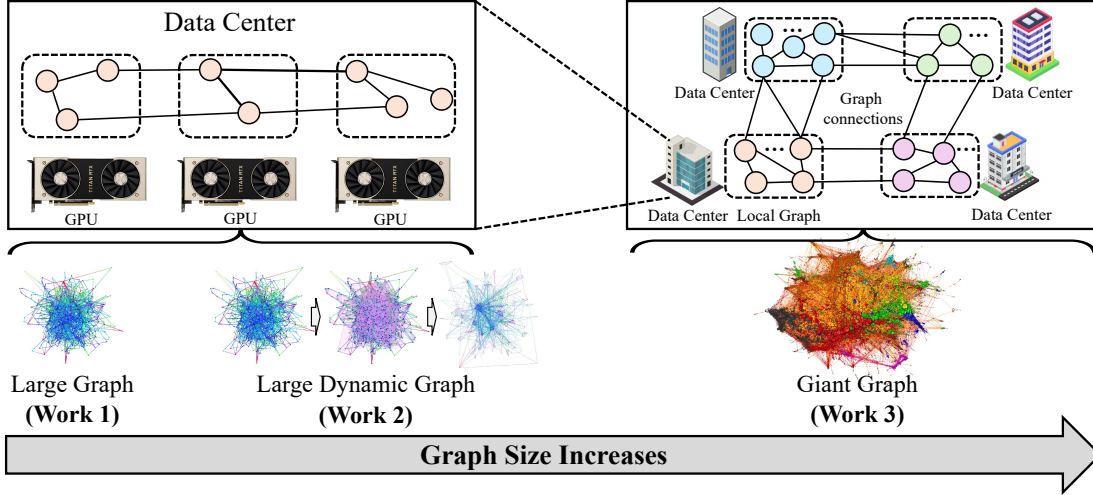


Figure 1.1: An overview of our three works on distributed graph learning.

own their local graphs. In this case, the local graph data cannot be shared among different data centers due to various privacy issues. Therefore, the main challenge here is how data centers collaboratively to train the graph model with their local graph data to improve the model performance while maintaining privacy-preserving. In addition, the local training process should be optimized to accelerate the whole collaborative training process. From the graph size perspective, although our three works all focus on distributed graph learning on large graphs, the studied graphs still have distinct differences in graph sizes. In our first work, we focus on distributed graph learning within a single data center on traditional static large graphs, e.g., social networks. Then, we study dynamic graphs, which are typically larger than traditional static graphs in our second work. Finally, we consider giant graphs, which are distributed across multiple data centers and cannot be trained by any single data center. Through our three works, we provide a comprehensive optimization solution for distributed graph learning.

In each work of this dissertation, we face unique challenges due to the differences in distributed scales and graph sizes. Specifically, we summary the main challenges in each work as follows.

- **Challenges in work 1.** In our first work, we consider the traditional static large graph, which will be trained within a single data center. Due to the large graph size, the data center will adopt multiple GPUs to accelerate the training process. However, the GPUs within the data center could be heterogeneous, complicating the optimization for resource utilization. For example, as the data center infrastructure grows and incorporates new GPUs, new GPUs must integrate effectively with existing ones to maximize resource utilization. Previous studies merely assume GPU homogeneity, which can lead to inefficient resource utilization when GPUs are heterogeneous. This heterogeneity can cause synchronization challenges during training, leading to underutilization of resources. Moreover, there could also be multiple different training jobs at the same time, exacerbating GPU resource competition. The main challenge in this setting is *how to accelerate distributed graph learning on heterogeneous GPUs while optimizing resource utilization*.

- 
- **Challenges in work 2.** Our second work focus on dynamic graphs, which can evolve (nodes and edges change) over time. The dynamic graphs, e.g., traffic graphs that describe real-time traffic flows of roads, are typically larger than traditional static graphs. We also consider that the dynamic graph is trained on multiple GPUs within the single data center. The most effective method for learning on dynamic graphs is Dynamic Graph Neural Network (DGNN). While there is significant interest in developing DGNN models, system-level support for such networks is less explored, introducing significant challenges for distributed training. Specifically, the most crucial challenge here is how to partition the dynamic graph among different GPUs to minimize cross-GPU traffic, which has also been recognized as the main system bottleneck by existing works. Yet, existing works adopt straightforward partitioning methods, which incurs significant communication costs among GPUs. Therefore, the main challenge here is *how to partition the dynamic graph among multiple GPUs to minimize cross-GPU communication while improving resource utilization*.
  - **Challenges in work 3.** In our final work, we move the giant graph, which is typically distributed among different data centers. The graph data within different data centers cannot be shared due to various privacy issues. A common example is medical records in hospitals, which can be organized as graphs where each node represents a patient record containing sensitive personal and health information. Each data center wants to use the whole graph information to train the GNN model so that the model performance can be maximized. Yet, GNNs typically require aggregating features from neighboring nodes to leverage the structural information of the graph, posing a significant challenge under privacy constraints. Moreover, the substantial size of the giant graph can result in significant training costs in each data center, reducing the efficiency of whole training process. The primary challenge here is *how to enable data centers to collaboratively train the GNN models efficiently on the giant graph while preserving privacy*.

This dissertation proposes a series of optimizations to address the above challenges in each work. We aim to develop a distributed graph learning platform that not only addresses the inherent challenges of large-scale, dynamic, and sensitive graph data but also maximizes the utilization of distributed computing resources. In summary, this dissertation makes the following contributions.

- **(Chapter 3)** Our first work introduces Hare for large graphs on heterogeneous GPUs. Hare not only improves the efficiency of distributed graph learning on heterogeneous GPUs but also tackles the complex scheduling problem inherent in managing multiple concurrent distributed learning jobs, each consisting of multiple training tasks on GPUs. Considering the GPU heterogeneity in training capabilities, Hare implements a relaxed fixed-scale synchronization approach, which allows tasks within the same job to process flexibly on different GPUs. This strategy not only improves resource utilization but also accommodates varying job demands and GPU performance. Hare then introduces a fast heuristic scheduling algorithm designed to minimize the total weighted job completion time, considering both job characteristics and GPU heterogeneity. Furthermore, Hare incorporates an advanced job switching technology to optimize the GPU execution environment during switching between different learning jobs, enhancing overall system efficiency.

- **(Chapter 4)** Our second work represents DGC for dynamic graphs. DGC innovates with a novel graph partitioning strategy that divides dynamic graphs into graph chunks, which are essentially subgraphs across spatial and temporal boundaries. This partitioning is achieved using a lightweight graph coarsening technique that ensures each chunk maintains a balanced workload and minimal inter-chunk edges. In addition, DGC further enhances the training efficiency with two key runtime optimizations: chunk fusion and adaptive stale embedding aggregation. Chunk fusion consolidates smaller graph chunks into larger ones before GPU loading, reducing unnecessary data handling and zero paddings. The adaptive stale embedding aggregation utilizes potential similar embeddings generated during different epochs, reducing inter-GPU communication and optimizing resource utilization without compromising the accuracy of the training process.
- **(Chapter 5)** Our final work shows FedGraph for giant graph training on multiple data centers. FedGraph mainly addresses the privacy concerns in giant graph training across different data centers while improving training efficiency. Specifically, FedGraph resolves the conflict between exploiting the complete graph information and privacy protection by designing a federated graph learning framework. This approach allows different data centers to enhance model performance by exchanging only the GNN models trained on their local graphs, rather than exchanging the sensitive graph data itself. In addition, FedGraph embeds node features into low-dimensional representations before the operation of neighbor aggregation so that original features cannot be recovered. To optimize local GNN training, FedGraph leverages Deep Reinforcement Learning (DRL) to develop an intelligent sampling algorithm, which dynamically adjusts neighbor sampling strategies to balance computational overhead with training accuracy.
- We comprehensively evaluate our works with careful system implementation and extensive experiments, demonstrating that our methods can significantly outperform existing work. First, given a set of heterogeneous GPUs and multiple concurrent distributed learning jobs, Hare can reduce the total job completion time by about  $2\times$ , compared to other scheduling methods. Second, for the distributed graph learning on dynamic graphs, DGC achieves a  $1.25\times$  -  $7.52\times$  speedup over the state-of-the-art training systems. Finally, besides strong privacy protection, FedGraph can further reduce the time to convergence by about  $2.5\times$  while achieving higher accuracy, compared to other sampling methods.

The rest of this dissertation is organized as follows. Chapter 2 gives the general background about graphs and graph learning. Chapter 3 presents the distributed graph learning system for large graphs on heterogeneous GPUs. An efficient distributed training system for dynamic graphs is given in Chapter 4, followed by the training system for giant graphs across different data centers in Chapter 5. Finally, Chapter 7 concludes this dissertation.

# Chapter 2

## Background

We here introduce some basic background about graphs and graph learning. We first introduce static graphs, which are typically learned by GNN. Then, we introduce dynamic graphs, which evolve over time. The most effective learning method on dynamic graphs is DGNN.

### 2.1 Graphs and Graph Neural Networks

There are an increasing number of applications and their data can be represented in the form of graphs. Formally, let  $G = (V, E)$  denote the graph, where  $V$  represents the set of nodes and  $E$  stands for the set of edges. Each node  $v \in V$  is associated with a feature vector  $X_v$  and a corresponding label  $y_v$ . Our primary objective of graph learning pertains to node classification. Specifically, our aim is to acquire a representation vector  $z_v$  for each node  $v$  that closely approximates its corresponding label  $y_v$ .

For node  $v$  in the  $k$ -th layer of the GNN model, it need to AGGREGATE and COMBINE the messages from its neighbors to update the representation. Formally, the  $k$ -th layer of a GNN is:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)} \left( \{h_n^{(k-1)} : n \in N(v)\} \right), \quad (2.1)$$

$$h_v^{(k)} = \text{COMBINE}^{(k)} \left( h_v^{(k-1)}, a_v^{(k)} \right), \quad (2.2)$$

where the initial state was defined as  $h_v^{(0)} = X_v$ , and  $N(v)$  denotes the set of neighbors of  $v$ . The choice of  $\text{AGGREGATE}^{(k)}(\cdot)$  and  $\text{COMBINE}^{(k)}(\cdot)$  can vary in different GNN models.

One of the most effective GNN models is Graph Convolutional Network (GCN) [21]. A GCN contains  $L$  convolutional layers, each of which has the same structure as the original graph  $G$ . In the  $l$ -th layer, each node  $v$  is represented by a vector  $h^{(l)}(v)$ , which is called node embedding. The first layer is the input graph and we have  $h^{(1)}(v) = x(v)$ . As shown in Fig.2.1, the graph convolution operation aggregates embeddings of neighboring nodes, transfers the results into low-dimensional representations, and finally feeds them to an activation function  $\sigma(\cdot)$ , e.g., ReLU, to generate node embeddings of the next layer. Formally, the propagation rule of GCN can be defined as follows:

$$Z^{(l+1)} = QH^{(l)}W^{(l)}; \quad H^{(l+1)} = \sigma(Z^{(l+1)}), \quad (2.3)$$



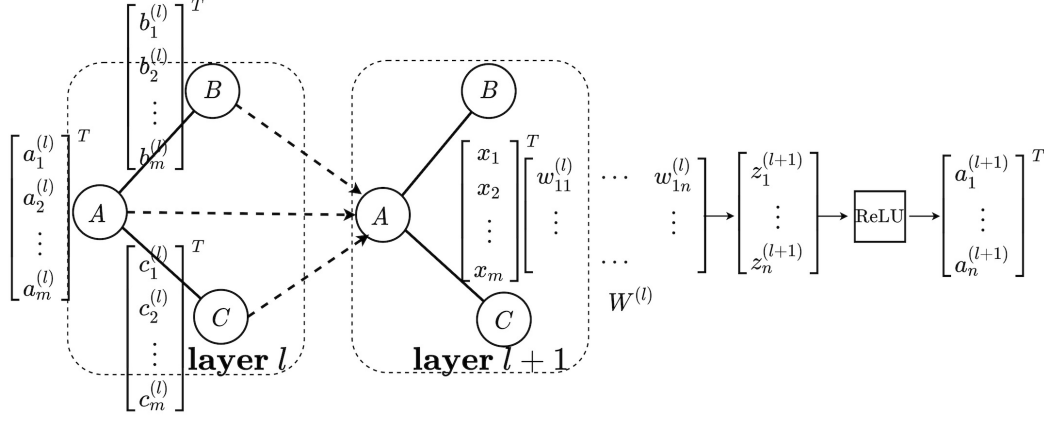


Figure 2.1: Illustration of graph convolution operation.

where  $H^{(l)}$  includes all node embeddings in the  $l$ -th layer, and  $Q = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$ . For the matrix  $\tilde{D}$ , we have  $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$  and  $\tilde{A} = A + I$ , where  $I$  is an identity matrix. The feature weights included in  $W^{(l)}$  are trainable parameters. Given some nodes with labels, we can train the feature weight matrix  $W^{(l)}$  using gradient descent algorithms. The trained parameters can be used to classify the nodes without labels.

**Optimization for GNNs.** Numerous studies have focused on scaling Graph Neural Networks (GNNs) for large graph training, which can be divided into two main categories. The first category focuses on algorithmic approaches, where existing works have explored various techniques to scale GNN models. These techniques include sampling methods [10, 22, 23], quantization [4, 5, 8], simplification [2, 3], and distillation [13–15]. The second category emphasizes distributed training, where GNN training is conducted using multiple CPUs or GPUs to manage large graphs [20, 24–34]. Several works have built upon general runtime frameworks, such as DGL [35], PyG [36], and AGL [37], to propose various optimizations. AliGraph [38] and AGL [37] only support distributed GNN training on CPUs, while others [24, 31, 39] support GNN training on GPUs. DistDGL [19] optimizes graph data access by supporting a distributed in-memory key-value store. DGCL [25] improves distributed GNN training efficiency with an efficient communication library and NVLink. Roc [16] minimizes data swapping between GPU memory and host DRAM by using dynamic programming.  $P^3$  [27] jointly combines intra-layer model parallelism and data parallelism to avoid communication costs for data-intensive node features among GPUs. Dorylus [33] deploys GNNs with serverless computing and increases training scalability at a low cost.

## 2.2 Dynamic Graphs and Dynamic Graph Neural Networks

A dynamic graph can be represented by  $\mathcal{G} = \{G_1, G_2, \dots, G_T\}$ , where  $G_t = (V_t, E_t)$  is a snapshot at timestep  $t$ .  $V_t$  and  $E_t$  represent the vertex and edge sets of snapshot  $G_t$ , respectively. Each vertex  $v_{i,t}$  in  $V_t$  is associated with a feature vector  $x_{i,t}$ . In addition, a vertex  $v_{i,t}$  has spatial and temporal neighbors. (1) *Spatial neighbors*, denoted as  $\mathcal{NS}(i, t)$ , are the vertices that are directly connected to  $v_{i,t}$  through an edge in the same snapshot  $G_t$ . They represent the immediate connections or relationships among

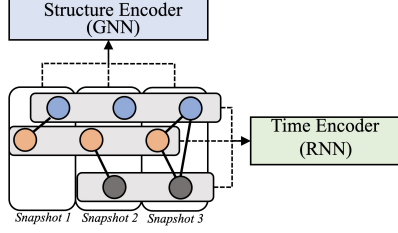


Figure 2.2: DGNN execution flow.

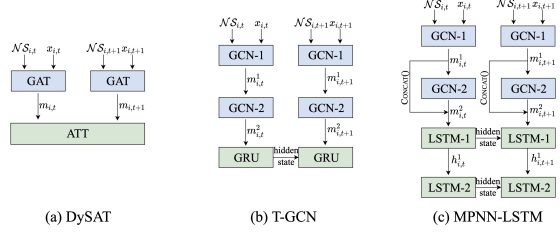


Figure 2.3: DGNN layer structures.

the vertices in a specific timestep. (2) *Temporal neighbors*, denoted as  $\mathcal{NT}(i, t)$ , are the vertices corresponding to the same entity as  $v_{i,t}$  but in different snapshots. They represent the changes or evolution of vertex features across different timesteps.

A dynamic graph neural network (DGNN) is composed of multiple blocks, where each block consists of a *structure encoder* and a *time encoder*, as illustrated in Fig. 2.2. The structure encoder extracts hidden information for each vertex by aggregating information from its structural neighbors. Meanwhile, the time encoder accumulates information for each vertex from its temporal neighbors. Note that different DGNN models have different implementation of structure and time encoders. For example, T-GCN [40] uses three 2-layer GCN [21] as the structure encoder, and a 1-layer GRU [41] model as the time encoder. DySAT [42] incorporates a 1-layer graph attention network (GAT) [43] and a 1-layer scaled dot-product attention model [44] within each of its DGNN blocks. Fig. 2.3 gives three representative DGNN models.

**Structure Encoder.** A graph neural network (GNN) is typically adopted to realize the structure encoder []. Snapshots are operated in the structure encoder independently. Consider a snapshot  $G_t = (V_t, E_t)$  associated with a adjacency matrix  $A_t$ , the  $l$ -th structure encoder layer takes the outputs of the last time encoder layer, denoted by  $h^l$ , where  $h^1 = X_t$ . The vertex embeddings are updated to  $z_t^l$  by:

$$z_t^l = \sigma(\hat{A}h^lW_{str}^l), \quad (2.4)$$

where  $\hat{A}$  is calculated from  $A$  and varies across GNN models used in the DGNN.  $W_{str}^l$  is the learnable weight matrix of the  $l$ -th structure encoder layer and  $\sigma$  is a activation function such as ReLU.

**Time Encoder.** In each layer of the time encoder, vertex sequences are processed independently, where a vertex sequence is a set of same vertices in different snapshots. Consider a vertex sequence  $(u_1, u_2, \dots, u_t)$ , the  $l$ -th time encoder layer calculates the temporal information based on the outputs  $z_u^l = (z_{u,1}^l, z_{u,2}^l, \dots, z_{u,t}^l)$  from the structure encoder by:

$$h_u^{l+1} = \text{Softmax}\left(\frac{QK^T}{\sqrt{d}} + \mathcal{M}\right)\mathcal{V}, \quad (2.5)$$

$$Q = z_u^l W_{tem}^{Q,l}, \quad K = z_u^l W_{tem}^{K,l}, \quad \mathcal{V} = z_u^l W_{tem}^{V,l}, \quad (2.6)$$

where  $W_{tem}^{Q,l}$ ,  $W_{tem}^{K,l}$ ,  $W_{tem}^{V,l}$  are learnable weight matrix in  $l$ -th time encoder layer and  $\sqrt{d}$  is a scaled parameter.  $\mathcal{M}$  is a  $t \times t$  mask matrix for different temporal encoding methods, which varies in different DGNN models [42, 45, 46].

**Optimization for DGNNs.** Two general frameworks, PyGT [47] and TGL [48], have been proposed to implement a variety of DGNN models. CacheG [49] improves DGNN

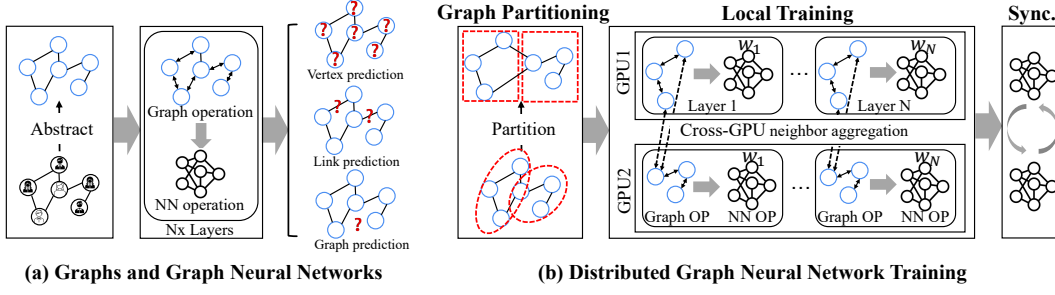


Figure 2.4: Distributed GNN training. (a) An example of Graphs and GNNs, where NN indicates the neural network. (b) Illustration of distributed GNN training, where  $W_i$  means the weights of  $i$ -th GNN layer.

training performance by introducing intermediate result caching. Cambricon-G [50] combines a dedicated architecture, featuring a cuboid engine and hybrid on-chip memory, to decrease energy consumption and on-chip memory access for dynamic GNNs. TGOpt [51] specifically targets attention-based DGNNs and introduces a range of optimizations, such as deduplication, memorization, and precomputation, to minimize redundant computation during DGNN inference. PiPAD [52] aims to enhance training efficiency and reduce data transfer overhead in the traditional “one-graph-at-a-time” DGNN training pattern. However, these works only support optimization for single-GPU DGNN training. In contrast, DGC focuses on efficient distributed DGNN training for handling large dynamic graphs.

## 2.3 Distributed Graph Learning

The workflow of traditional distributed graph learning can be divided into three stages that are graph partition, local GNN model training, and model synchronization. Figure 2.4 visualizes the high-level abstraction of end-to-end distributed GNN training workflow.

**Graph partitioning.** This stage is a crucial preprocessing step that facilitates distributed training by partitioning the graph structure and its features into subgraphs, which are then distributed among a set of workers, such as GPUs. Unlike traditional distributed machine learning, the training data in GNNs are dependent, incurring additional complexity to the data partitioning stage. As illustrated in Figure 2.4, the edges between subgraphs indicate data dependencies. Recognizing these dependencies can reduce the efficiency of distributed training due to increased communication. Conversely, ignoring these dependencies can compromise model accuracy. Therefore, the graph partitioning process is vital for efficient end-to-end distributed GNN training. Since GNNs involve graph computation tasks, traditional graph partition methods like those used in systems such as AliGraph [20] and DistGNN [26], including METIS [53], Vertex-cut [54], Edge-cut [53], and streaming graph partitioning [55], can be applied directly to GNN workloads. Recent empirical studies by [56] on the performance of various graph partitioning dimensions (1-D, 2-D, 1.5-D, and 3-D) with respect to GCN models have shown that classical methods may not optimally balance GNN workloads while minimizing communication costs.

After partitioning, each worker undertakes a batch generation operation if mini-

batch training is enabled. Each worker generates a computation graph from their assigned subgraph. However, due to the inherent data dependencies, generating computation graphs for mini-batch training in GNNs is different from traditional deep learning models. The necessity to access remote input data complicates the distributed batch generation process more than in traditional models.

**Local training.** The model execution is specifically designed for distributed GNN training. The model training can be further divided into the execution model and communication protocol [57]. For distributed mini-batch training only the GNN model is involved, which manages the scheduling of different training operators and how to overlap them efficiently with the batch generation operation. For distributed full-graph training, both the training model and communication protocol is included. The execution model involves  $k$ -layer graph aggregation of GNN models and the aggregation exhibits an irregular data access pattern. Furthermore, the graph aggregation in each layer needs to access the hidden embeddings and computed gradients of the remote neighbors via the communication protocol, and the synchronization schema between layers should be also considered.

**Synchronization.** The synchronization stage is consistent with that of traditional DNN model training. Specifically, the existing techniques in classical distributed machine learning can be directly applied to distributed GNN training. In conclusion, the distributed GNN model training stage is more complicated than traditional DNN training and needs careful design for both the execution model and the communication protocol.

## 2.4 Graph Partitioning

Partitioning graphs across multiple GPUs is essential to minimize cross-GPU traffic during distributed graph training. The graph partitioning problem has been extensively studied in distributed GNN training. For example, DistDGL [19], AliGraph [20], and DistGNN [26] adopts the Metis partitioning algorithm [58] to optimize cross-GPU communication costs. 1.5D, 2D, and 3D graph partitioning are adopted in [56] to improve distributed GNN training. NeuGraph [24] adopts the Kernighan-Lin algorithm and Roc [16] uses a linear-regression based algorithm to partition graphs.  $P^3$  [27] independently partitions the input graph and features to avoid communicating huge features over the network. However, these graph partitioning methods do not apply to dynamic graph partitioning since they are designed for unraveling spatial dependency. Recently, several works have been proposed for dynamic graph partitioning in distributed DGNN training. DynaGraph [59] partitions the dynamic graph by temporal sequences, which effectively eliminates temporal embedding transmissions. Chakaravarthy et al. [60] propose a joint partitioning method that applies Partitioning by Spatial Snapshots (PSS) to assign snapshots to GPUs to execute structure encoders, and then shuffling to Partitioning by Temporal Sequences (PTS) for running time encoders. However, existing partitioning methods may not be suitable for various datasets, as they do not account for spatio-temporal non-uniformity in dynamic graphs. In contrast, DGC introduces a partitioning method based on graph chunks that takes full advantage of spatio-temporal non-uniformity in dynamic graphs. This approach leads to better workload balancing and reduced communication costs, significantly improving the DGNN training efficiency.

## 2.5 Job Scheduling

Job scheduling, which determines when and where each job should run, is the most fundamental and critical issue for distributed machine learning. Early studies follow the idea of traditional batch job scheduling by treating each job as an unsplittable unit and schedule them on different GPUs [61]. Later, some works have exploited the intra-job parallelism, i.e., tasks in the same training round of a job can run in parallel, which can significantly enhance learning performance. Optimus [62] allocates resources to ML jobs by learning a throughput model with respect to various resource allocation. Themis [63] introduces a notation of finish-time fairness to promote fair allocation, while improving the cluster utilization. Pollux [64] studies different resource allocation for ML jobs by observing the throughput and statistical efficiency during training. Zhang et al. [65] design an online algorithm that selects the amount of resources for each job to minimize the total job completion time. Although the above works have exploited both inter-job and intra-job parallelism, they consider homogeneous GPUs and forbid GPU preemption during job execution.

Recently, GPU-heterogeneity becomes popular as the expansion of data centers and it has attracted significant research attention. Gandiva<sub>fair</sub> [66] proposes an automated trading mechanism to support time-slicing resource sharing among different jobs while improving the cluster efficiency. Gavel [67] develops a heterogeneity-aware scheduler to generate different scheduling policies for different kinds of jobs. However, Gandiva<sub>fair</sub> and Gavel schedule jobs based on given time slice length. Such a coarse-grained scheduling manner leaves a large optimization space for performance improvement. Moreover, they ignore the task switching cost. Allox [68] transforms the job scheduling problem into a min-cost bipartite matching to provide dynamic fair allocation, but it conducts job-level scheduling and ignores the intra-job parallelism.

## 2.6 Federated Learning

The goal of federated learning is to train a shared model among distributed devices while avoiding the exposure of their training data. A typical federated setting consists of a number of devices, each of which holds a dataset that cannot be exposed to others. In addition, there is a parameter server responsible for synchronizing training results among devices. Federated learning contains multiple training rounds. In each training round, devices first download the latest global model from the parameter server and independently conduct training using their local data. Then, they send updated models or model differences back to the parameter server. After collecting training results from all devices, the parameter server integrates them to create a new global model. During the whole training process, devices share only models and it is almost impossible to infer the training data from these models. Due to the protection for training data, federated learning becomes one of the hottest topics in recent years and many important research efforts have been made to address various challenges [69–72]. Zhao et al. [70] have demonstrated the impact of non-IID data in federated learning with mathematical and proposed an approach that sends a set of uniform distribution data to each client to reduce the effect of non-IID data.

Recently, several works study GNNs under different federated settings from the one in this work. Suzumura et al., [73] develop a federated learning platform to detect financial crime activities across multiple financial institutions. They extract global graph

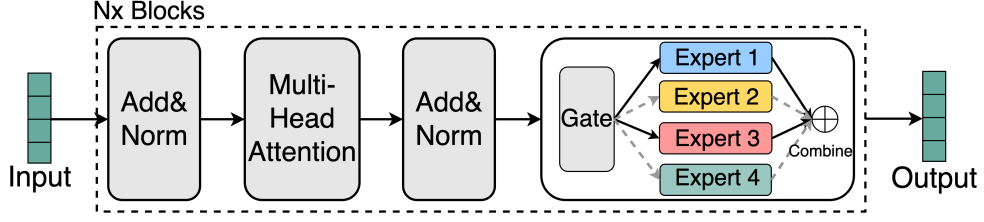


Figure 2.5: The illustration of the MoE architectures.

information to Euclidean data by graph analytic methods instead of graph neural networks. Besides, they assume the global graph belongs to all clients. In contrast, we study GCNs on non-Euclidean data, and each client owns a local graph.

Jiang et al., [74] propose a novel distributed surveillance system based on GNN and federated learning. There are two critical differences between this work and our work. First, they consider a cross-device federated setting, involving a large number of cameras with limited computation and communication capability. In contrast, we study a cross-silo federated setting, which typically involves a small number of clients. Second, they aim to protect the trained model. However, we explore inter-client connections and protect node features.

Mei et al., [75] study federated privacy-preserving graph neural networks with a vertical federated setting, i.e., assume that graph structural, features, and labels belong to different sources. However, we consider a horizontal federated setting, i.e., each local client maintains a complete graph dataset with its own graph structure, node features, and labels.

## 2.7 MoE Architecture

The potential of the MoE architecture to improve accuracy, while achieving sub-linear scaling in computational costs, has been validated by existing works [76, 77]. The MoE architecture replaces the Feed-Forward Network (FFN) layer within a Transformer block into a MoE layer, where a Transformer model typically consists of multiple blocks. As illustrated in Figure 2.5, the MoE layer includes a gate network and multiple experts. For every input token<sup>1</sup>, the gate network assigns itself to several specific experts according to token-to-expert assignment scores.

## 2.8 Offloading-based model serving.

Existing works introduce offloading computationally intensive tasks to more capable servers [79]. For example, Luo et al. [?] suggest offloading data, such as videos captured by UAVs, to remote cloud servers for processing, in applications related to disaster management. Lynskey et al. [80] propose a system where UAVs offload images, coupled with an optimization framework for managing the computational resources dedicated to processing these offloaded tasks. Valentino et al. [81] develop a mechanism that allows UAVs to offload computational tasks to nearby UAVs. Liu et al. [82] introduce a

<sup>1</sup>Data is commonly called *tokens* in natural language processing. For computer vision, tokens are typically pixels or patches [78].

cooperative offloading approach between UAVs, employing a deep reinforcement learning (DRL) methodology. Yang et al. [83] propose a hierarchical architecture that take partial execution of Convolutional Neural Network (CNN) model inference on UAVs, with the remaining portion processed at edge servers. Dai et al. [84] explore a vehicle-assisted offloading architecture for UAVs, establishing a matching scheme based on UAVs' preference lists for vehicles. Alessio et al. [85] implement offloading for UAVs using reinforcement learning (RL), offering an effective formalization to minimize both the requisite information and training time for the RL agent. Hu et al. [86] present a pipeline framework enabling the operation of large-scale models across multiple edge devices. Xu et al. [87] approach the problem by decomposing a sizable Vision Transformer (ViT) model to facilitate collaborative processing by multiple edge devices. The offloading methods for UAVs, however, introduce significant privacy concerns when sending data to public servers. Moreover, the inference requests will be delayed by the overhead of sending data to and receiving results from remote servers.

## 2.9 Tiny ML-based model serving.

Recent advancements in tiny machine learning (ML) technologies, such as quantization, pruning, and knowledge distillation, have been proposed to enable local inference on UAVs. These technologies reduce the model size and resource requirements, thereby eliminating the need to transmit data to public servers. For instance, Ringwald et al. [?] enhance the speed of object detection on UAVs through an automatic model pruning approach. Similarly, Wang et al. [88] apply a model pruning method based on principal component analysis for real-time fault detection on UAVs. The Yolo-LiteDense model, derived from a densely structured Yolo-V3 model through channel pruning, is utilized for detection tasks on UAVs [89]. Bultmann et al. [90] implement a lightweight CNN model on UAVs for efficient object detection, while Vandersteegen et al. [91] achieve fast object detection using layer fusion and lower-bit quantization techniques. Additionally, Fouda et al. [92] introduce a lightweight hierarchical framework that adaptively switches between simple and advanced models for early forest fire detection via UAVs. Li et al. [93] introduce a block-structured pruning method aimed at reducing the storage and computational demands of large-scale models, while Rahman et al. [94] employ quantization, transforming the model into an optimized FlatBuffer format to enhance efficiency. However, these tiny ML technologies often entail a trade-off between model size reduction and inference accuracy drop, which are used for conventional models, such as CNNs. In the context of giant models, the resource demands for inference significantly surpass the capabilities of UAVs, which needs the scaling down of model sizes by several orders of magnitude. Traditional tiny ML techniques struggle to accomplish without compromising on high inference accuracy. Thus, achieving efficient inference serving of giant models on UAVs while maintaining robust accuracy remains a challenging endeavor.

## Chapter 3

# Efficient Distributed Graph Learning on Heterogeneous GPUs

### 3.1 Problem Statement

The recent success of machine learning, especially deep learning, stems from the availability of big data and strong computational power brought by cutting-edge hardware (e.g., GPUs and TPUs). Facing massive computational loads, it is inefficient or sometimes impossible to train models on a single GPU, driving attention towards distributed machine learning on multiple GPUs. In the paradigm of Distributed Machine Learning (DML), a learning job is divided into multiple tasks, which can run on multiple GPUs in parallel. The Parameter Server (PS) [95] scheme has been widely adopted to coordinate the training processes across multiple GPUs.

In practice, it is rare to assign a dedicated GPU cluster to each DML job, due to low resource utilization [63]. Instead, a common practice is to let multiple jobs share these GPUs. A critical research challenge is how to efficiently schedule these jobs on GPUs, which is particularly concerned by public or private cloud data centers that offer learning services while desiring high hardware resource utilization. Therefore, the learning job scheduling problem has attracted great research attention, and various solutions have been recently proposed with different objectives. For example, Gandiva [96] has studied GPU sharing among several jobs to improve GPU utilization. The fairness of learning jobs has been studied by Pollux [64]. Zhang et al. [65] have exploited both intra-job and inter-job parallelism and proposed online DML job scheduling algorithms to minimize job completion time.

However, the above works are all based on an assumption that GPUs are homogeneous. In practice, hardware heterogeneity commonly exists in computing clusters. For example, as the expansion of data centers, new GPUs are continuously added and they should work with existing ones to maximize resource utilization. Some recent works [66–68] have started to pay attention to the influence of GPU-heterogeneity, which motivates us to re-examine the DML job scheduling problem in such an emerging heterogeneous computing environment. We find that existing works with the homogeneity assumption cannot fully achieve their claimed goals in heterogeneous environment. That is because they expect that training tasks scheduled simultaneously on several machines have the same completion time. However, when these tasks actually run on heterogeneous machines, they complete at different time. Due to the task synchronization at the end of each training round, the faster tasks need to wait for slower



ones, which may lead to longer job completion time.

Hardware heterogeneity brings new challenges as well as opportunities to DML system design. We are excited to see the success of several preliminary studies. For example, Gandiva<sub>fair</sub> [66] is designed to ensure the user-level fairness while maximizing the efficiency of heterogeneous GPU clusters. Gavel [67] generalizes existing scheduling policies with consideration of GPU heterogeneity. Allox [68] efficiently schedules DML jobs in a heterogeneous cluster to improve the max-min fairness. These recent works have extensively studied inter-job parallelism in heterogeneous computing environment, but leaving intra-job parallelism unexplored. They treat each DML job as a unsplitable unit when making scheduling decisions. We are still facing open questions: how to exploit both inter-job and intra-job parallelism on heterogeneous GPUs? How much acceleration can be obtained? And is there strong theoretical support for such acceleration?

In this work, we propose Hare for heterogeneous GPU cluster scheduling, to answer the above questions. The basic idea of Hare can be illustrated using the example in Fig. 3.1. There are 3 jobs, and every job consists of several tasks, each of which responsible for training a data batch. We further assume that job J3 needs to synchronize for every two tasks. The single-batch training time on 3 different GPUs is shown in the table. By following the heterogeneity-oblivious strategies in [65], we get the scheduling results shown in Fig. 3.1(a), where the job J3 uses GPU2 and GPU3 to exploit intra-job parallelism and J2 takes the whole GPU1. When both jobs complete, we start J1 that runs on two GPUs in parallel. The total job completion time is 10.5 seconds and the makespan is 4.5 seconds. The results of job-level scheduling aware of GPU-heterogeneity, represented by Allox [68], are shown in Fig. 3.1(b). Each job gets a dedicated GPU and the total completion time of 9 seconds. An alternative scheduling result with better performance is shown in Fig. 3.1(c), where the idle time on GPU3 can be used by J1. This scheme reduces total job completion time to 8.5 seconds and makespan to 3 seconds. Note that although communication time and task switching overhead is ignored in this example for simplicity, we have similar observation even if they are considered.

Despite the promise of GPU-heterogeneity-awareness and intra-job parallelism, Hare needs to conquer several critical technical challenges to grasp the promised benefits. First, to use the idle time on some GPUs before model synchronization, as shown in Fig. 3.1(c), the scheduler needs to allow GPU preemption during learning job execution, which is forbidden by existing works [64, 65, 96]. Moreover, with such GPU preemption, switching between tasks belonging to different jobs should be quick. Otherwise, frequent task switching may happen in Hare, leading to ruinous overhead. To solve this challenge, Hare first enables fast task switching by optimizing task initialization and cleaning on GPUs, which has been identified as the major source of switching overhead. We use some methods, e.g., CUDA context sharing, that have been shown to be effective in reducing task switching overhead [97]. Moreover, we exploit the unique features of Hare to further improve performance by proposing early task cleaning and speculative memory management.

Second, existing intra-job synchronization schemes are not flexible enough, which constrains the optimization space of Hare. Synchronization schemes decide how many independent tasks are launched in a training round and how to synchronize these tasks. Two synchronization schemes have been widely adopted. A scale-fixed scheme always launch the same number of tasks in each training round and schedule them when the

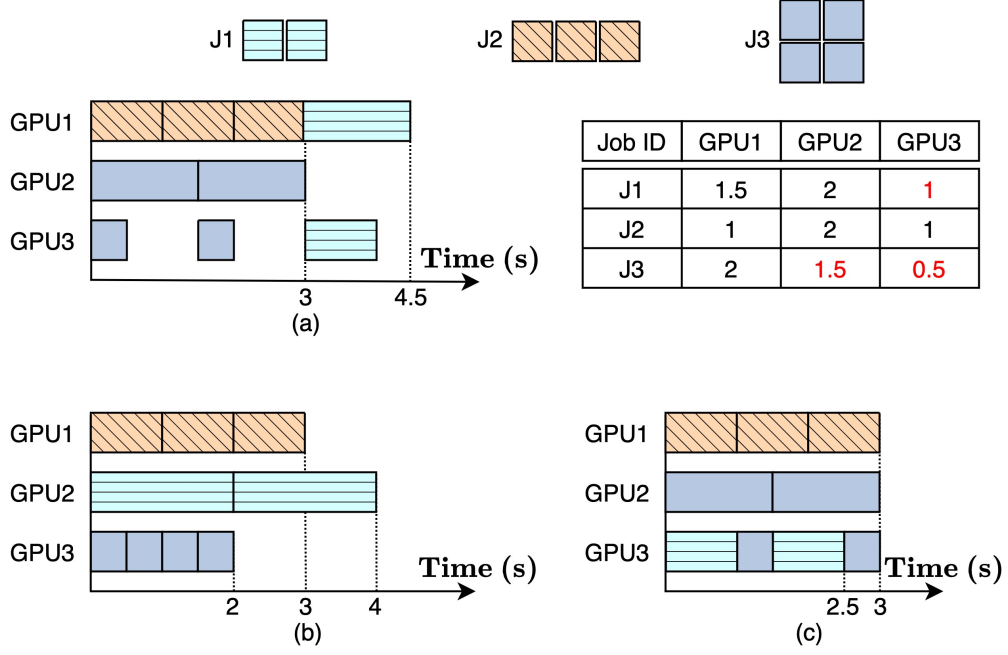


Figure 3.1: A toy example to show job scheduling results under different methods. (a) GPU-heterogeneity-oblivious scheduling result; (b) GPU-heterogeneity-aware scheduling result, but without exploiting intra-job parallelism; (c) A better scheduling result jointly considering GPU heterogeneity and intra-job parallelism.

same number of GPUs are available to maximize parallelism. Instead, a scale-adaptive scheme can adjust the number of parallel tasks according to available GPU resources. Although scale-adaptive scheme is more flexible but may lead to uncertainty in convergence. Comparison details of both schemes can be found in Section 3.2. Motivated by their pros and cons, we propose a relaxed scale-fixed synchronization scheme for Hare to maximize scheduling flexibility. It fixes the number of tasks in each rounds but relaxes resource requirement for scheduling, so that we can maintain convergence certainty while maximizing GPU utilization.

The final challenge is about scheduling algorithm design. We need to exploit the parallelism at both intra-job and inter-job levels while considering GPU heterogeneity. Different jobs may prefer different GPUs because their models and training datasets are diverse. A sophisticated scheduler should make careful decisions to optimize the overall performance. Thanks to our proposed fast task switching mechanism, the switching overhead is so tiny that we can ignore it in the scheduling algorithm design for simplicity. Even though, the scheduling problem is still NP-hard. We propose a fast heuristic algorithm to minimize the total weighted job completion time and derive an important theoretical approximation ratio to the optimal solutions.

We develop a prototype of Hare based on PyTorch 1.8.1 by adding about 2500 LoC of Python and C++. We build a testbed consisting of 15 heterogeneous GPUs (8 V100s, 4 T4s, 1 K80, and 2 M60s) for performance evaluation. The workloads contain 8 types of jobs, which train different deep learning models respectively. The details of models and datasets are shown in Table 3.2. We consider 4 scheduling algorithms proposed by recent works as comparison baselines.

We also develop a simulator using Python to evaluate Hare under large-scale set-

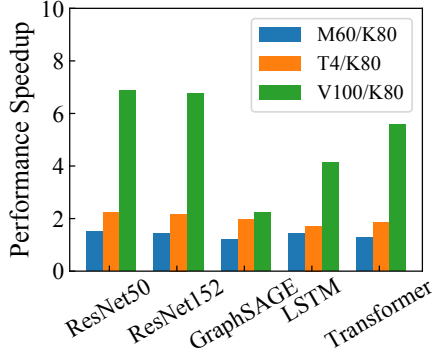


Figure 3.2: Training speedup of different jobs on different GPUs.

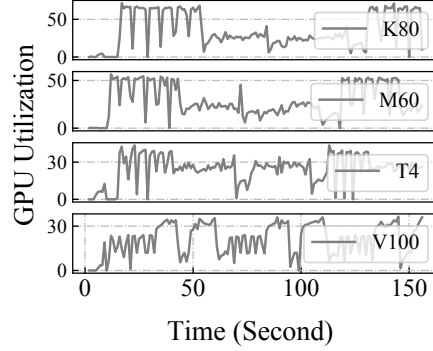


Figure 3.3: The GPU utilization of training GraphSAGE model.

tings. The accuracy of our simulator has been verified by comparing its results with the one obtained from the testbed. They have no more than 5% difference. We collect tasks' running traces from our testbed and synthesize large-scale traces to feed this simulator. The results show that Hare outperforms baselines under various settings.

Since Hare currently uses an offline scheduling algorithm, it is short in handling dynamic jobs. This dynamic comes from two kinds of cases. First, jobs may change settings (e.g., hyper-parameters or parallelism levels) during running. For example, some jobs use the autoML technique to search for the best models over different hyper-parameters or even model structures. Hare needs to frequently profile task running time, which may cause high overhead. Second, jobs arrive in different time and we cannot accurately predict future job arrivals. Online algorithms are needed to address the dynamic in both cases.

Despite these limitations, we believe this work still have important and sufficient contributions. We have strong theoretical results for offline scheduling. A basic scheduling system has been built and it can be easily extended to accommodate other scheduling algorithms for dynamic jobs, which is left for future work.

## 3.2 Motivation

### 3.2.1 GPU heterogeneity and inter-job parallelism.

We find that different GPUs provide different performance speedups for learning jobs, mainly because of the heterogeneity of model (such as model architecture) and hardware. As shown in Fig. 3.2, we use the training time per mini-batch on a K80 GPU as the baseline and evaluate the speedup for other GPUs. Training the ResNet50 model can be sped up by 2x on a T4 GPU, while with 7x significant speedup on a V100 GPU. However, the graph learning model GraphSAGE shows the heterogeneous performance on different GPUs. Specifically, GraphSAGE can only be sped up by about 2x, even on the most advanced V100 GPU. That is because the required FLOPS of GraphSAGE are much smaller than other models. Moreover, the data pre-processing speed is slower than the GPU computation speed. The GPU spends more time to wait for input data, resulting in low GPU utilization. As shown in Fig. 3.3, we find that utilization of GPU is less than 30% when we train GraphSAGE on a V100 GPU. There is a little improvement when training GraphSAGE on a V100 GPU. Therefore, giving a high

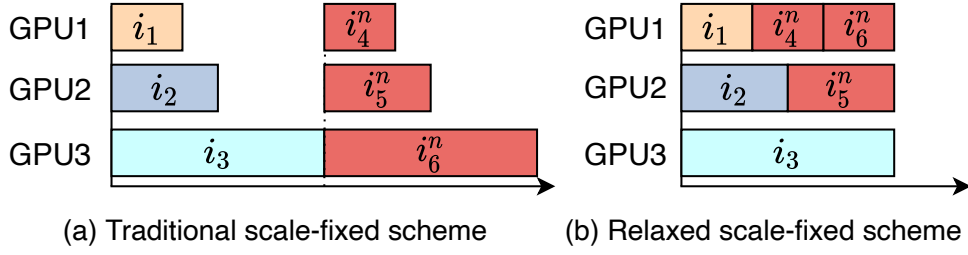


Figure 3.4: An example showing the benefit of relaxed scale-fixed synchronization scheme adopted by Hare.

priority for assigning V100 GPUs to the ResNet50 job is more efficient since it shows a high-performance speedup than other jobs.

This empirical study gives us important hints about accelerating learning jobs and increasing GPU utilization. On the other hand, it throws challenges about how to schedule jobs on GPUs, considering massive learning workloads and hardware resources in modern data centers. Moreover, the intra-job parallelism, which will be presented in the following, further complicates this problem.

### 3.2.2 GPU heterogeneity and intra-job parallelism.

Each DML job consists of multiple tasks, which are periodically synchronized to share gradients, via a parameter server or exchanging gradients directly. Although a single advanced GPU can provide a performance speedup for gradients computing, the training speed of the whole DML jobs is constrained by the synchronization. We train the ResNet152 on five different distributed settings and show the epoch time in Fig. 3.5. We find that mixing different GPUs is not always helpful. For example, compared to a pure K80 cluster, adding faster T4 or V100 brings no acceleration. That is because the gradient synchronization impedes early completed GPUs to move to the next-round training. There is much idle time on V100 GPUs when they wait for the gradients update from K80 GPUs. This low efficiency can be also reflected by GPU utilization as shown in Fig. 3.6, where we can see that K80 is always busy while V100’s utilization is rarely over 50%.

A straightforward idea to address this challenge is to schedule parallel tasks belonging to the same job on similar GPUs. However, it is hardly to have such a perfect allocation in practice because of limited GPU resources in the cluster. Since it is inevitable to use heterogeneous GPUs for intra-job parallelism, it is desired an algorithm that can well schedule fine-grained tasks to reduce idle time.

### 3.2.3 Scale-fixed synchronization versus scale-adaptive synchronization

Existing intra-job parallelism methods can be categorized into two types, scale-fixed and scale-adaptive, according to how many tasks are synchronized. Scale-fixed methods, adopted by Tiresias [98] and Gandiva [96], fix the number of synchronized tasks and always try to allocate the same number of GPUs so that they can achieve full parallelism. If the number of available GPUs is insufficient, all tasks need to wait until

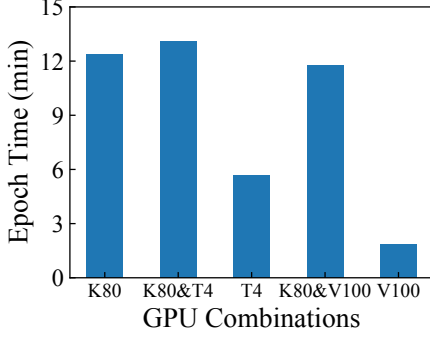


Figure 3.5: Epoch time of ResNet152 under different GPU combinations.

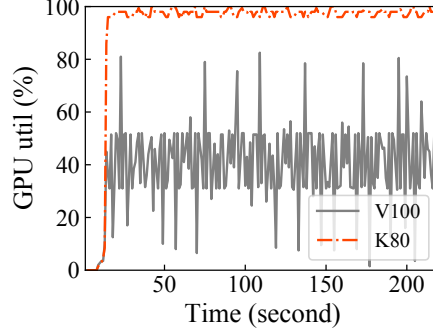


Figure 3.6: GPU utilization of V100 and K80 when training ResNet152.

required GPU number is satisfied. In contrast, scale-adaptive methods [62, 63, 67, 99] dynamically change the number of synchronized tasks according to available GPU resources. Although these methods are flexible and tasks are not blocked by strict resource requirement, we may need more training epochs to achieve competitive accuracy of scale-fixed methods. Moreover, it is hard to build theories to predict how many epochs are needed. Due to this uncertainty, we do not use scale-adaptive design in Hare.

Motivated by the above analysis, we would like to follow the scale-fixed idea but relax the parallelism requirement. An example is shown in Fig. 3.4, where three tasks,  $i_1$ ,  $i_2$  and  $i_3$ , are running on 3 GPUs respectively. Now a new job  $n$  consisting of 3 tasks (i.e., synchronization scale is 3) comes. As illustrated in Fig. 3.4(a), traditional scale-fixed methods start job  $n$  after the completion of slowest task  $i_3$ , when 3 GPUs are available. We find that it is unnecessary to make 3 tasks strictly run in parallel. Two tasks can run sequentially on GPU1, as shown in Fig. 3.4(b), leading to earlier completion than traditional methods while maintaining the same level of parallelism.

Implementing such a relaxed scale-fixed synchronization method is not easy. We need to address challenge of changing the task assignment and synchronization modules. It also affects task scheduling algorithm design.

### 3.2.4 Task Switching Cost

As we are motivated above, exploiting intra-job parallelism on heterogeneous GPU environment is critical for accelerating training and increasing hardware resource utilization. It should be taken into consideration of scheduling algorithm design. We further find that such an algorithm inevitable generates results with frequent task switching on GPUs. To study task switching cost on GPUs, we conduct experiments to compare switching time and task time under 3 different settings. In the first setting, we alternatively run a GraphSAGE task and a ResNet50 task, each of which trains a mini-batch. We define a metric  $\Omega = \frac{t_{sw}}{t_c^g + t_c^r}$  to evaluate the switching cost, where  $t_{sw}$  is the task switching time,  $t_c^g$  and  $t_c^r$  is the average batch training time of GraphSAGE and ResNet50, respectively. As shown in Fig. 3.7, the switching cost is about 9 times higher than training. Similar high cost can be observed under other two settings. We also show the real-time GPU utilization with and without task switching in Fig. 3.8. When training a single ResNet50 model on a V100 GPU, GPU resources are almost fully utilized. However, if we train GraphSAGE and ResNet50 alternatively, GPU utilization is no more than 50%, because much time is spent on CUDA environment cleaning and cre-

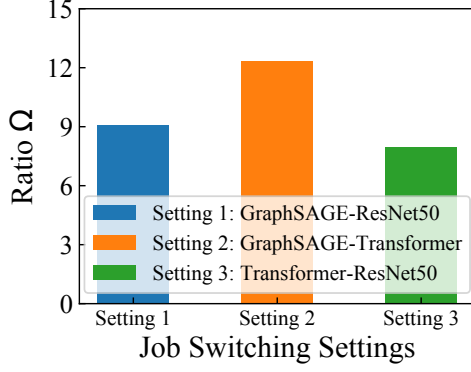


Figure 3.7: Ratio of switching time and training time under three settings.

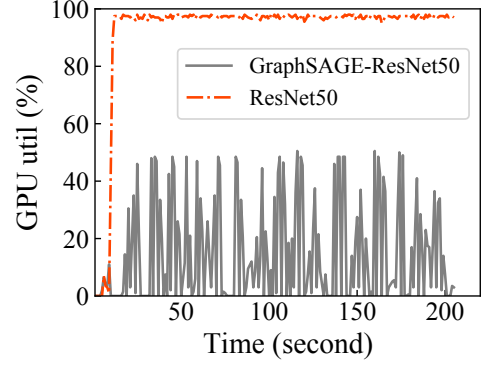


Figure 3.8: V100 GPU utilization with and without task switching.

ation during task switching.

Even though we can continuously schedule the same-type tasks on a GPU, which is possible due to the proposed relaxed scale-fixed method, to amortize switching cost. However, the switching cost is still high and such a solution heavily relies on sophisticated design of scheduling algorithm that takes switching cost into consideration. In this work, instead of struggling on amortizing switching cost, we propose to reduce it using novel system designs.

### 3.3 System Overview

In this section, we give an overview about Hare’s design. First, we clarify three primary design goals of Hare.

- **High training efficiency:** Given a number of DML jobs, Hare needs to schedule them on a cluster of GPUs to finish the training as fast as possible. It depends on exploiting both intra-job and inter-job parallelism as well as GPU heterogeneity, as we shown in the motivation section. We achieve this goal by designing a high-efficient task scheduling algorithm with strong theoretical performance guarantee.
- **High GPU utilization:** Hardware utilization is always important for cluster providers. Hare is not only a job scheduler but also a resource manager in the GPU cluster. Therefore, Hare aims to improve the GPU utilization by reducing system cost and minimizing GPU idle time.
- **Starvation-free:** In real scenarios, learning tasks can not wait for arbitrarily long time or starve due to mutual exclusion resource requirement. The scheduling algorithm design should be starvation-free so that every task has a chance to run.

A system overview of Hare is shown in Fig. 6.3, where Hare is integrated into the existing PS-based distributed machine learning framework. Hare is not only a scheduling algorithm, but also a set of modules that optimize training processes across GPUs. It contains two main components: a logically centralized task scheduler, and executors running on training machines. All data are stored with HDFS [100]. The whole system running process contains two stages, an offline preparation stage and an online



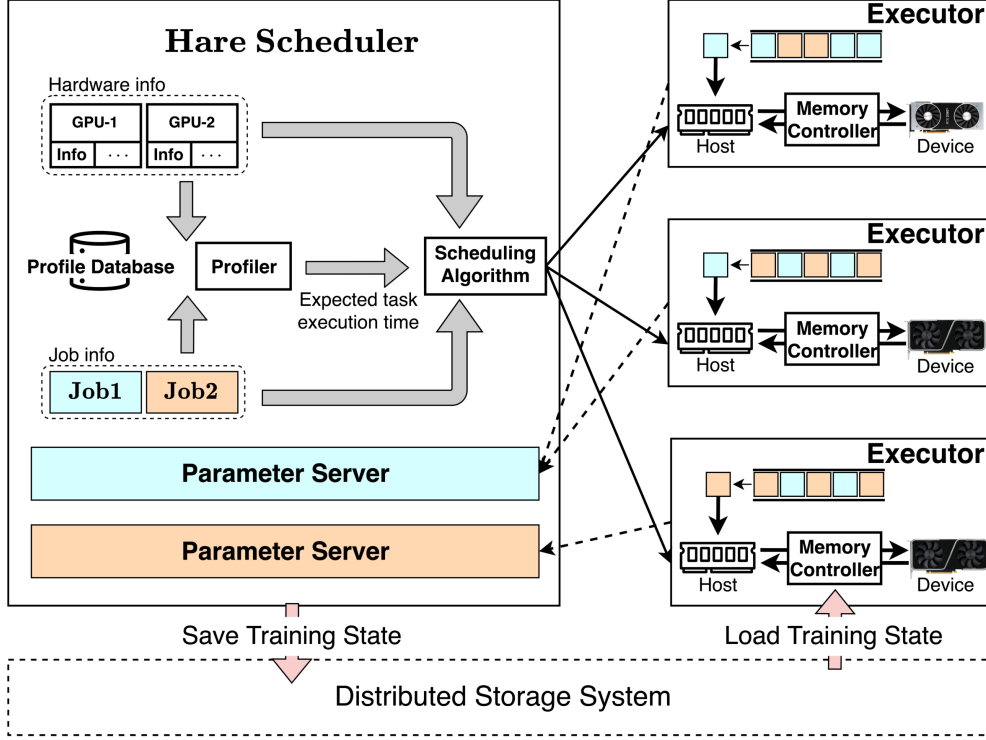


Figure 3.9: System Overview.

training stage. In the preparation stage, the task scheduler is fed by job information, e.g., job types, model description and training data size, from upper-layer applications. It also collects hardware information, e.g., GPU types, speed and memory, from the under-layer computing infrastructure. These information first goes to a module called profiler that trains a small piece of data to obtain expected task execution time on different GPUs, which will be the input of the task scheduling algorithm. We note that some jobs are usually repeatedly submitted to the training platform. For example, some models are periodically re-trained using latest collected datasets to adapt to emerging cases, which is particularly common in deep reinforcement learning. This observation motivates us to accelerate the profiling by maintaining a database that stores historical profiling results. We first search the database upon receiving job information. If corresponding results can be found, we skip profile training and directly feed searching results to the scheduling algorithm. We then run the scheduling algorithm (in Section 3.5) to generate a task running sequence for each GPU. Finally, these task sequences are sent to corresponding executors.

In the training stage, each executor schedules tasks and loads checkpoints on GPU according to their order given in the received task sequence. When a task completes, it sends updated gradients to the corresponding parameter server for aggregation. We follow the most of training designs in traditional distributed machine learning frameworks [62, 63], except the task switching mechanism. In existing works, since each job has exclusive use of assigned GPUs, several consecutive tasks on a GPU belongs to the same job and they share the same GPU context, leading to low switching overhead. In contrast, Hare allows GPU preemption by alternatively running tasks of different jobs, which involves frequent context switching with high overhead. Therefore, we design a fast task switching mechanism with a customized memory controller (in Section 3.4) to

reduce this overhead.

## 3.4 Fast Task Switching

Given two tasks scheduled continuously on a GPU, a traditional task switching process contains two main steps. First, the predecessor needs to clean its GPU environment by freeing GPU memory occupations. Second, the successor initializes its environment by creating a new CUDA context, launching the process, allocating GPU memory and moving the model from main memory to GPU memory. Traditionally, the above two steps run sequentially, which incurs large overhead. Such overhead has been observed by [97, 101, 102] and our experimental results have also confirmed it. Therefore, reducing the task switching cost becomes a critical challenge that has to be addressed by Hare.

The fast task switching of Hare is mainly motivated by PipeSwitch [97]. To accelerate the model movement from main memory to GPU memory, PipeSwitch leverages the layered structure of neural networks and pipelines the model transmission and execution. Moreover, it finds that CUDA context creation is slow, and proposes to create multiple CUDA contexts in advance to hide the overhead of context creation during task switching. Other techniques, like NVIDIA Multiple Process Sharing (MPS) [101], allow multiple processes to share a single GPU, but they cannot be applied here because these processes need to be pre-loaded into GPU memory, which may exceed GPU memory limit.

Similar designs have been also adopted by PipeSwitch [97]. However, it is originally designed for maximizing GPU utilization by filling GPU idle time of an inference job with training or other inference jobs, and it misses many optimization chances in training job scheduling scenarios studied in this work. To further reduce task switching cost, we propose the following two new designs by exploiting unique features of our task scheduling problem.

**Early Task cleaning.** When a task completes, PipeSwitch cleans its GPU environment by deleting GPU memory pointers only, leaving memory content unmodified, which may cause security issues [103, 104]. For example, an attacker can steal the private data by allocating the memory region the same as that used by the previous task. Instead, we propose early task cleaning that deletes intermediate data of each layer once its backward training completes. Early task cleaning has two benefits. First, we delete not only memory pointers but also memory content to avoid potential security concerns. Second, released GPU memory can be used for pre-loading data of the next task, so that it can start earlier.

**Speculative Memory Management.** PipeSwitch cleans a task by removing all its data stored in GPU memory. However, we find that it is not always necessary, especially when we know the tasks that will be scheduled on the same GPU. As an example shown in Fig. 3.10, we consider three tasks,  $i_1$ ,  $i_2$  and  $i_3$ , scheduled sequentially on a GPU. Tasks  $i_1$  and  $i_3$  belong to the same training round of a job, while  $i_2$  is from a different job. We further suppose that three tasks do not occupy the whole GPU memory, which is common in practice [105]. In a traditional design, all data of  $i_1$  are removed when it completes. An alternative method adopted by Hare is to keep the model data of task  $i_1$ , so that they can be re-used by  $i_3$  scheduled later. Such a speculative memory management is feasible because Hare conducts an offline task scheduling and task



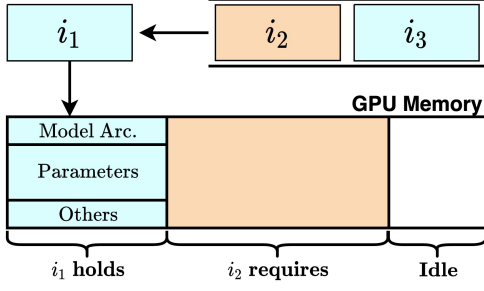


Figure 3.10: An example of speculative memory management.

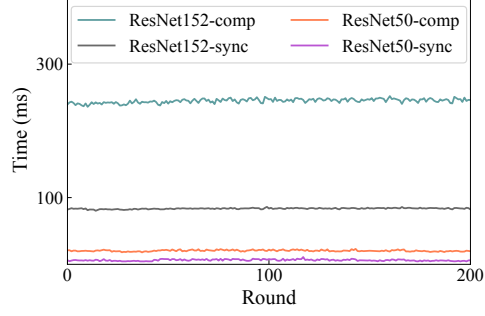


Figure 3.11: Training two popular models on 8 V100 GPUs.

running sequences as well as their GPU memory occupation can be known in advance. To decide which models and how long they can be kept in GPU memory, Hare uses a simple heuristic that always gives higher GPU memory priority to the next tasks, and greedily keeps models of latest completed tasks until they cannot be accommodated. Of course, we can formulate this memory management problem as an optimization problem and solve it to get the optimal solution. However, we find that the heuristic works sufficiently well in practice, and the resulted switching cost can be neglected.

## 3.5 Task Scheduling Algorithm

In this section, we present the task scheduling algorithm, which is the core design of Hare. The system model is first presented, followed by algorithm details and theoretical performance analysis.

### 3.5.1 System Model

We consider a set  $\mathcal{N}$  of training jobs, running on a set  $\mathcal{M}$  of heterogeneous GPUs. Each job  $n \in \mathcal{N}$  consists of multiple training rounds, which are denoted by set  $R_n$ . Each job  $n \in \mathcal{N}$  launches a set  $D_r$  of training tasks that can run in parallel in every training round, and each task is responsible for training a data batch. After local training, all tasks synchronize their gradients via the PS scheme to obtain an updated model for the next-round training. Moreover, we denote  $\mathcal{D}$  as the set of all tasks in  $\mathcal{N}$ .

We let  $T_{i,m,r}^c$  denote the training time of task  $i$  on GPU  $m$  in the training round  $r$ , and the corresponding synchronization time is denoted by  $T_{i,m,r}^s$ . As shown in Fig. 3.11, our experimental results about 2 popular models have shown that task training time and synchronization time is highly predictable and stable across training rounds. This fact allows us to use  $T_{i,m}^c$  and  $T_{i,m}^s$  by dropping the subscript  $r$  to simplify problem formulation. More importantly, it makes task scheduling with performance guarantee feasible.

Due to GPU heterogeneity, each task may have different training time on different GPUs. Similarly, it may have different synchronization time across GPUs because network condition changes. Besides, we assume that the training time is longer than the synchronization time. That is because GPUs are usually connected by high-speed networks (e.g., NVLink and InfiniBand) in data centers. Note that this is different from

Table 3.1: Notations

$\mathcal{N}$	set of training jobs	$\mathcal{M}$	set of heterogeneous GPUs
$a_n$	arrive time of job $n$	$w_n$	weight of job $n$
$R_n$	set of training rounds for job $n \in \mathcal{N}$		
$D_r$	set of parallel tasks in $r \in R_n$		
$\mathbf{D}$	set of all tasks in $\mathcal{N}$		
$T_{i,m,r}^c$	training time of task $i \in \mathcal{N}$ on GPU $m$ in the training round $r$		
$T_{i,m,r}^s$	synchronization time of task $i \in \mathcal{N}$ on GPU $m$ in the training round $r$		
$x_i$	the start time of task $i \in \mathcal{N}$		
$y_{i,m}$	whether task $i \in \mathcal{N}$ is assigned to GPU $m$		
$\hat{x}_i$	the solution of $x_i$ from the relaxed problem		
$\tilde{x}_i$	the solution of $x_i$ from Algorithm 1		
$\tilde{y}_{i,m}$	the solution of $y_{i,m}$ from Algorithm 1		
$H_{i,m}$	the middle completion time of task $i \in \mathcal{N}$ on GPU $m$		
$H_i$	the maximum middle completion time of task $i \in \mathcal{N}$		
$\boldsymbol{\pi}$	a non-descending sequence according to $H_i$		
$\varphi_m$	current available time of GPU $m$		

job-level non-preemption assumed in existing works [62, 96, 98], i.e., a job  $n \in \mathcal{N}$  cannot be preempted once it starts to run on GPUs.

We consider the non-preemption setting for task running, i.e., a task's execution cannot be preempted once it is scheduled on a GPU. Thanks to the fast task switching mechanism, there is tiny task switching cost, which is less than 5% of task training time according to experimental results. Therefore, we ignore the task switching cost in the problem formulation for simplicity. Note that our main theoretical results are still true even this cost is considered. We list the important notations in Table. 6.1

Each job  $n \in \mathcal{N}$  is associated with arrive time  $a_n$  and a weight  $w_n$ . To formulate the problem, we define a variable  $x_i$  to denote the start running time of task  $i$ . In addition, a binary variable  $y_{i,m}$  is defined to indicate the GPU assignment, i.e.,  $y_{i,m} = 1$  if task  $i$  is assigned to GPU  $m$ , and  $y_{i,m} = 0$  otherwise. The completion time of job  $n$  is denoted by  $C_n$ . With the objective of minimizing the total weighted job completion time, we formulate the task scheduling problem as follows:

$$\mathbf{Hare\_Sched:} \quad \min \sum_{n \in \mathcal{N}} w_n C_n, \quad \text{subject to:}$$

$$x_i \geq a_n, \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \quad (3.1)$$

$$\sum_{m \in \mathcal{M}} y_{i,m} = 1, \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \quad (3.2)$$

$$C_n \geq x_i + \sum_{m \in \mathcal{M}} y_{i,m} (T_{i,m}^c + T_{i,m}^s), \quad \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \quad (3.3)$$

$$x_j \geq x_i + \sum_{m \in \mathcal{M}} y_{i,m} (T_{i,m}^c + T_{i,m}^s), \forall j \in D_{r+1}, i \in D_r, \quad r \in R_n, n \in \mathcal{N}; \quad (3.4)$$

$$|x_i - x_j| \geq y_{k,m} T_{k,m}^c, \forall i, j \in \mathcal{D}, m \in \mathcal{M}, \quad y_{i,m} = y_{j,m} = 1, k =_{k=\{i,j\}} \{x_k\}. \quad (3.5)$$

Constraint (3.1) indicates that tasks of each job can not start before arrival. Each task can be assigned to at most one GPU, which is represented by (3.2). The job completion time is constrained by (3.3). Besides, due to the synchronized parameter update policy, the tasks of the  $(r + 1)$ -th round must wait for the completion of all tasks of the  $r$ -th round, as shown in (3.4). Finally, we use constraint (3.5) to guarantee the non-preemption among tasks. Specifically, for any two tasks assigned to the same GPU, one cannot start before the completion of the other. The hardness of the above problem is given as follows.

Hare\_Sched is NP-hard. The NP-hardness of Hare\_Sched can be proved by reducing the well-known SS13 [106] problem. The details are ignored due to length limit.

### 3.5.2 Algorithm Design

By carefully examining the problem formulation, we find that the difficulty mainly stems from the non-linear constraint (3.5). Therefore, we are motivated to relax (3.5) and then design a heuristic algorithm based on the solution of the relaxed problem. This algorithm contains the following two steps and pseudo codes are shown in Algorithm 1.

**Step 1: Problem Relaxation.** We relax the original planning problem as follows:

$$\mathbf{Hare\_Sched\_RL:} \quad \min \sum_{n \in \mathcal{N}} w_n C_n, \quad \text{subject to:}$$

$$\sum_{i \in \mathcal{D}} y_{i,m} T_{i,m}^c (x_i + T_{i,m}^c) \geq \frac{1}{2} \left[ \left( \sum_{i \in \mathcal{D}} y_{i,m} T_{i,m}^c \right)^2 + \sum_{i \in \mathcal{D}} \left( y_{i,m} T_{i,m}^c \right)^2 \right], \forall m \in \mathcal{M}; \quad (3.6)$$

(3.1) – (3.4).

Constraint (3.6) is the relaxation of (3.5) according to [107]. Note that (3.6) is in-

**Algorithm 1** Task Scheduling Algorithm in Hare

---

```

1:  $\tilde{x}_i = 0, \tilde{y}_{i,m} = 0, \forall i \in \mathcal{D}, m \in \mathcal{M}$ ;
2:  $\varphi_m = 0, \forall m \in \mathcal{M}$ ;
3: Solve the Hare_Sched_RL problem to obtain solutions  $\hat{x}_i$  as well as  $H_i$ ;
4: Sort tasks to generate a sequence  $\pi$  satisfying  $H_{\pi(1)} \leq H_{\pi(2)} \leq \dots \leq H_{\pi(|\mathcal{D}|)}$ ;
5: for  $i \in \{\pi(1), \pi(2), \dots, \pi(|\mathcal{D}|)\}$  do
6:   Identify the corresponding job  $n$  and  $r$  where  $i \in D_r, r \in R_n$ ;
7:   if  $r = 0$  then
8:      $t_i = a_n$ ;
9:   else
10:     $t_i = \max_{j \in D_{r-1}} \{\tilde{x}_j + \tilde{T}_j^c + \tilde{T}_j^s\}$ ;
11:   end if
12:   Find  $m^* = \arg\max_{m \in \mathcal{M}} \varphi_m$ ;
13:    $\tilde{x}_i = \max\{t_i, \varphi_{m^*}\}$ ;
14:    $\tilde{y}_{i,m^*} = 1$ ;
15:    $\tilde{T}_i^c = \tilde{y}_{i,m^*} T_{i,m^*}^c, \tilde{T}_i^s = \tilde{y}_{i,m^*} T_{i,m^*}^s$ ;
16:    $\varphi_{m^*} = \tilde{x}_i + \tilde{T}_i^c$ ;
17: end for
18: return  $\tilde{x}, \tilde{y}$ 

```

---

dependent of the running order of tasks scheduled on each GPU and it always holds for any feasible scheduling. Thus, Hare\_Sched\_RL serves as a lower bound of Hare\_Sched. Although Hare\_Sched\_RL is a mixed-integer quadratic programming that is still with high theoretical complexity, we are fortunate to have fast solvers, e.g., CPLEX and Gurobi, which have been well optimized and work well in practice.

**Step 2: Task Scheduling.** The solution of Hare\_Sched\_RL is denoted by  $\hat{x}_i$ . The middle completion time of task  $i$  on GPU  $m$  can be calculated by  $H_{i,m} = \hat{x}_i + \frac{1}{2}T_{i,m}^c$ . Therefore, the maximum middle completion time of task  $i$  can be denoted by  $H_i = \max_m \{H_{i,m}\}$ .

We then sort tasks in a non-descending order according to  $H_i$  to generate a sequence  $\pi$ , as shown in line 2. The  $j$ -th task in the sequence  $\pi$  is denoted by  $\pi(j)$ . Next, we iteratively schedule tasks according to the sequence  $\pi$  in the **for** loop in lines 5-17. Specifically, in each iteration, we deal with the task  $i$  by first identifying its associated job  $n$  and training round  $r$ , which is easy in practice by attaching such information in the task description. In the following line 7, we continue to check whether this task belongs to the first training round. If it is, i.e.,  $r = 0$ , this task can logically start upon arrival. We let  $t_i$  denote the task available time. In this case, we have  $t_i = a_n$  as shown in line 8, where  $a_n$  is the arrival time of job  $n$  containing task  $i$ . Otherwise, task  $i$  needs to wait the completion of all tasks in the previous training round, and its available time  $t_i$  can be calculated in line 10. Note that  $\tilde{T}_j^c$  and  $\tilde{T}_j^s$  in line 10 is the real training and synchronization time of tasks in the  $(r - 1)$ -th round and they are updated in line 15 in the previous algorithm iterations.

Up to now, we consider only task available time, which may not be identical to its real start time in practice because we haven't decided which GPU to run this task. We let  $\varphi_m$  denote the current GPU available time. Next, we will check the GPU availability and find a GPU where the task can be assigned. We adopt a greedy strategy, which

always assigns the task to the GPU  $m^*$  with the earliest available time, as shown in line 12. After that, we can update the real task start time, denoted by  $\tilde{x}_i$ , as well as the task assignment  $\tilde{y}_{i,m^*}$ . The real task training time and synchronization time is updated in line 15. Finally, we update the GPU available time  $\varphi_{m^*}$  in line 16. Note that the synchronization time  $T_{i,m^*}^s$  is not considered in line 16 because the communication can overlap with the next task assigned on this GPU.

### 3.5.3 Theoretical Analysis

Before deriving the approximation ratio of the proposed algorithm, we prove the following two lemmas. Due to the space limitation, we put the complete proof in our technical report [108].

For any task  $\pi(j) \in \pi$  on GPU  $m$ , we have:

$$\sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \leq 2H_{\pi(j)}, \quad (3.7)$$

where  $\tilde{y}_{\pi(k),m}$  indicates the GPU assignment by Algorithm 1.

For any task  $\pi(j)$  as well as its predecessors  $\{\pi(1), \pi(2), \dots, \pi(j-1)\}$  in the sequence  $\pi$ , constraint (3.6) always holds and we have:

$$\begin{aligned} \sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c (\tilde{x}_{\pi(k)} + T_{\pi(k),m}^c) &\geq \\ \frac{1}{2} \left[ \left( \sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \right)^2 + \sum_{k=1}^j (\tilde{y}_{\pi(k),m} T_{\pi(k),m}^c)^2 \right]. \end{aligned} \quad (3.8)$$

By substituting  $H_{i,m} = \hat{x}_i + \frac{1}{2}T_{i,m}^c$  and eliminating  $\frac{1}{2} \sum_{k=1}^j (\tilde{y}_{\pi(k),m} T_{\pi(k),m}^c)^2$  in the right side of (3.8), we can obtain:

$$\sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c H_{\pi(k),m} \geq \frac{1}{2} \left( \sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \right)^2. \quad (3.9)$$

Because of  $H_{\pi(k)} = \max_m \{H_{\pi(k),m}\}$  and  $H_{\pi(1)} \leq H_{\pi(2)} \leq \dots \leq H_{\pi(j)}$ , we have:

$$H_{\pi(j)} \sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \geq \frac{1}{2} \left( \sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \right)^2. \quad (3.10)$$

Canceling out  $\sum_{k=1}^j \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c$  in both sides of (3.10) leads to (3.7).

We let  $\pi_m$  denote the task sequence on GPU  $m$  returned by Algorithm 1. The total idle time before task  $\pi_m(j)$  on GPU  $m$  is  $\delta(\pi_m(j), m)$ , which satisfies:

$$\delta(\pi_m(j), m) \leq \alpha H_{\pi_m(j)}, \quad (3.11)$$

where  $\alpha = \max_{i \in \mathcal{D}} \{T_i^{c,max}/T_i^{c,min}, T_i^{s,max}/T_i^{s,min}\}$ .

Without loss of generality, we consider two tasks  $\pi_m(j-1)$  and  $\pi_m(j)$ , which are not continuously scheduled on GPU  $m$ . Suppose task  $\pi_m(j)$  belongs to the job  $n$ . Let us first check how such a case happens. According to Algorithm 1, when we schedule task  $\pi_m(j)$  according to the task sequence  $\pi$ , GPU  $m$  has the earliest available time. However, task  $\pi_m(j)$  cannot start immediately after  $\pi_m(j-1)$  because of the synchronization barrier, i.e., there must exist some tasks, which belong to previous rounds of  $\pi_m(j)$ , running on other GPUs before the start of task  $\pi_m(j)$ . Otherwise, there would be no GPU idle time between  $\pi_m(j-1)$  and  $\pi_m(j)$ . Note that there may be multiple training rounds of job  $n$  between tasks  $\pi_m(j-1)$  and  $\pi_m(j)$ . In each round, there must be a bottleneck task whose running time is the same with duration of this round. Otherwise, some tasks would be scheduled on GPU  $m$  between tasks  $\pi_m(j-1)$  and  $\pi_m(j)$ . We denote these bottleneck tasks as  $\langle u(1), u(2), \dots, u(l) \rangle$ . We have the following relationship between tasks  $\pi_m(j-1)$  and  $u(0)$ .

$$\tilde{x}_{u(0)} \leq \tilde{x}_{\pi_m(j-1)} + \tilde{T}_{\pi_m(j-1)}^c; \quad (3.12)$$

$$H_{u(0)} \geq H_{\pi_m(j-1)}. \quad (3.13)$$

where  $u(0)$  is a task that belongs to the previous round of  $u(1)$  and satisfies  $\tilde{x}_{u(0)} + \tilde{T}_{u(0)}^c + \tilde{T}_{u(0)}^s = \tilde{x}_{u(1)}$ . By introducing  $\alpha = \max_{i \in \mathcal{D}} \{T_i^{c,max}/T_i^{c,min}, T_i^{s,max}/T_i^{s,min}\}$  and adopting constraint (3.4), we can obtain:

$$\alpha(H_{\pi_m(j)} - H_{u(0)}) \geq \sum_{i=0}^l (T_{u(i)}^{c,max} + T_{u(i)}^{s,max}). \quad (3.14)$$

According to the property of the results  $\{\tilde{x}_i\}$  returned by the Algorithm 1:

$$\tilde{x}_{\pi_m(j)} - \tilde{x}_{u(l)} = \tilde{T}_{u(l)}^c + \tilde{T}_{u(l)}^s; \quad (3.15)$$

$$\tilde{x}_{u(i)} - \tilde{x}_{u(i-1)} = \tilde{T}_{u(i-1)}^c + \tilde{T}_{u(i-1)}^s, \forall i = 1, 2, \dots, l. \quad (3.16)$$

we final proof that:

$$\delta(\pi_m(j)) \leq \alpha(H_{\pi_m(j)} - H_{\pi_m(0)}) = \alpha H_{\pi_m(j)}. \quad (3.17)$$

Algorithm 1 is  $\alpha(2 + \alpha)$ -approximation. Since our algorithm always schedules tasks on machines with the earliest start time, we have:

$$\tilde{x}_{\pi(j)} \leq \sum_{k=1}^{j-1} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c + \delta(\pi(j), m), \forall m \in \mathcal{M}. \quad (3.18)$$

By combining with the result of adding (3.7) and (3.11), we obtain:

$$\tilde{x}_{\pi(j)} + \tilde{T}_{\pi(j)}^c \leq (2 + \alpha) H_{\pi(j)}. \quad (3.19)$$

We can proof Theorem 3.5.3 by taking scaling methods in (3.19).

## 3.6 Implementation

We have implemented a prototype of Hare by using roughly 2500 LoC of Python and C++. We use PyTorch 1.8.1 for DML job training. Hare primary maintains two components: a central scheduler and executors. The scheduler communicates with executors via controlling messages implemented using gRPC APIs [109].

**Scheduler.** The scheduler integrates a task profiler, a scheduling algorithm, and parameter servers. The scheduler first executes `task_profile()`, fed by job information, to predict task execution time. The task scheduling is executed to obtain the task sequence for each executor. According to job information, the scheduler instantiates a series of `Hare_Parameter_Server` to bind to each DML job for gradient synchronization. `Hare_Parameter_Server` saves the checkpoint of DML job by using PyTorch interface `save()`. Also, the scheduler maintains a gRPC module to communicate the task sequence and gradients with executors.

**Executor.** The executor initializes several trainer processes to train tasks in the sequence received from the scheduler. In our implementation, we initialize three trainer processes in the executor. To hide CUDA context creation cost, we create a CUDA context for each process in advance by calling an implicit initialization `torch.randn(10, device='cuda')`. When a task needs to be trained, we assign it to a process (called the working process) and leave the rest on standby. Each working process initializes the task model locally and loads the checkpoint from storage by using PyTorch interface `load()`. Note that the model structure is small so that we can save it locally. We add hooks to the model to enable pipelined model transmission. Specifically, we use PyTorch interface `register_forward_pre_hook()` to change the initialized property of components (e.g., `torch.nn.Linear()`) in each layer. After that, the executor starts the training task and sends gradients to the parameter server using PyTorch and gRPC interfaces. We also add hooks to change the property `retain_graph` of all tensors in each layer, which aims to support early task cleaning. In native PyTorch, the gradients of intermediate variables are deleted. To avoid the deletion, we modify the gradients free mechanism in PyTorch. Moreover, we keep the model data in GPU according to switching algorithm results.

## 3.7 Evaluation

In this section, we first introduce our experimental settings and then present the results of the testbed and simulations.

### 3.7.1 Experimental settings

**Testbed.** We build a testbed consisting of 15 heterogeneous GPUs (8 V100s, 4 T4s, 1 K80, and 2 M60s), which are deployed on 4 Amazon EC2 instances. All GPUs are equipped with PCIe-3×16 (15.75 GB/s). Each instance is powered by NVIDIA driver 418.21, CUDA 10.1 and cuDNN 8.0.4, running Ubuntu 18.04 with Linux kernel version 5.4. All instances are connected via the 25 Gbps Ethernet.

**Simulator.** We have developed a trace-driven simulator to evaluate Hare in large-scale settings. The simulator is built in Python, and emulates the execution of DML jobs using the traces collected from the testbed. The job arrival time is set according to the trace in Google cluster [123].

Table 3.2: Deep Learning Jobs Used in Our Experiments.

Type	Model	Dataset	BatchSize	
CV	VGG-19 [110]	Cifar10 [111]	128	25%
CV	ResNet50 [112]	Cifar100 [113]	64	
CV	Inception V3 [114]	Cifar100 [113]	32	
NLP	Bert_base [115]	SQuAD [116]	32	25%
NLP	Transformer [44]	WMT16 [117]	128	
Speech	DeepSpeech [118]	ComVoice [119]	8	25%
Rec.	FastGCN [120]	Cora [121]	128	25%
Rec.	GraphSAGE [122]	Cora [121]	16	

Table 3.3: Average Task Switching Time of Different Jobs.

	VGG19	ResNet50	Inception V3	Bert_base	Transformer	DeepSpeech	FastGCN	GraphSAGE
Default	3288.94 ms (98.21%)	5961.16ms (97.37%)	7807.43 ms (96.99%)	9016.99 ms (93.95%)	5257.17 ms (95.41%)	5125.64 ms (94.15%)	5327.24 ms (98.47%)	5213.54 ms (98.29%)
PipeSwitch [97]	4.01 ms (2.40%)	4.75 ms (5.46%)	5.03 ms (2.39%)	12.57 ms (1.99%)	10.34 ms (2.03%)	8.91 ms (1.59%)	2.86 ms (7.56%)	2.42 ms (8.64%)
Hare	2.77 ms (1.82%)	2.04 ms (3.71%)	2.46 ms (1.43%)	5.03 ms (1.13%)	5.79 ms (1.36%)	4.27 ms (1.25%)	1.83 ms (4.53%)	0.96 ms (3.36%)

**Workload.** We create some DML jobs based on 8 popular models across domains of computer vision (CV), natural language processing (NLP), speech, and recognition (Rec.). The details of these models are shown in Table 3.2. In the default setting, each type of jobs accounts for 25% of the total workload. All jobs are implemented in PyTorch 1.8.1, and they are trained using synchronous PS scheme. Since the original datasets of SQuAD and WMT16 are too large and the corresponding training would run for days, we downscale them so that they can complete within hours.

**Schemes for Comparison.** We compare Hare with following schemes.

*Gavel\_FIFO*: FIFO (First In First Out) is a default job scheduling algorithm in many traditional batch job processing systems [61]. It schedules jobs in an order according to their arrival time. Gavel [67] customizes FIFO for heterogeneous GPUs by assigning jobs to fastest available GPUs. If the number of idle GPUs is insufficient, this job needs to wait until demanded GPUs are available.

*SRTF (Shortest Remaining Time First)*: SRTF has been widely adopted to minimize total job completion time. It always schedule jobs that could complete earlier.

*Sched\_Homo* [65]: We denote a recent scheduling algorithm [65] designed for homogeneous GPUs by Sched\_Homo. Similar to Hare, it aims to minimize the weighted ML job completion time by exploiting both inter-job and intra-job parallelism. However, job-level preemption is not allowed.

*Sched\_Allox* [68]: We consider the ML job scheduling algorithm proposed by Allox [68]. The GPU heterogeneity has been fully exploited, but it does not consider the intra-job parallelism.



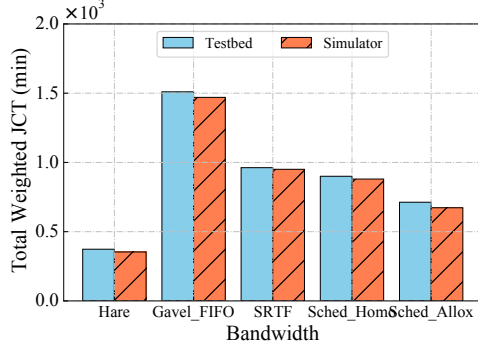


Figure 3.12: The results in testbed.

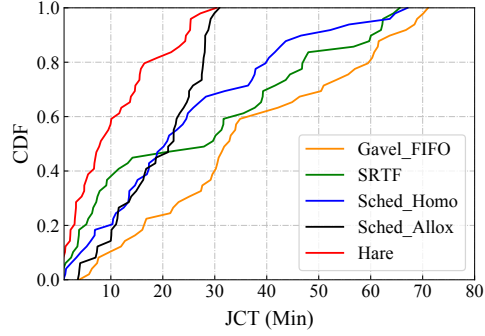


Figure 3.13: CDF of job completion time.

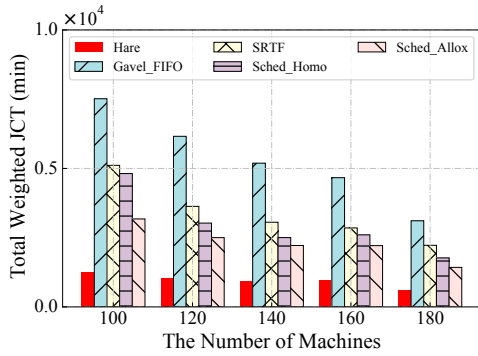


Figure 3.14: Performance under different number of GPUs.

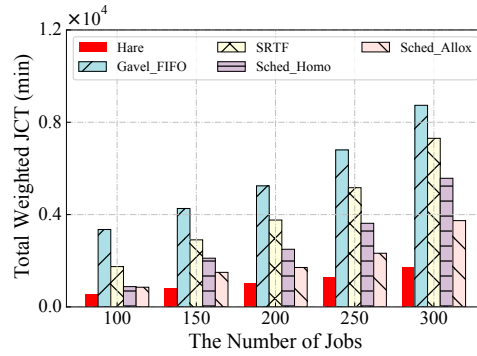


Figure 3.15: Performance under different number of jobs.

### 3.7.2 Results on Testbed

We first study the benefits of fast task switching by showing the average switching time of different jobs in Table 3.3. A default task switching scheme, without any optimization, needs more than 3000ms for all jobs. PipeSwitch can reduce the average switching time to 12.57ms for Bert\_base and less for others. The maximum switching time of Hare is no more than 6ms. The proportion of task switching time to the total task time is also shown in the table. We can see that Hare constrains the task switching overhead within 2% for most of models, and the largest overhead under FastGCN is no more than 5%. These results justify our assumption that task switching time is negligible in the scheduling algorithm design.

The total weighted job completion time (JCT) of several schemes running on the testbed and the simulator is shown in Fig. 3.12. Compared with other schemes, Hare can reduce total weighted JCT by 47.6% to 75.3%, significantly outperforming other schemes. Fig. 3.13 shows the cumulative distributed function (CDF) of JCT of all jobs. We can see that about 90.5% of jobs can complete within 25 minutes by Hare, while Sched\_Allox and Sched\_Homo can complete only 66.7% and 56.5%, respectively. That is because Allox misses the optimization chances brought by intra-job parallelism, and Sched\_Homo is GPU-heterogeneity-oblivious, leading to low GPU utilization.

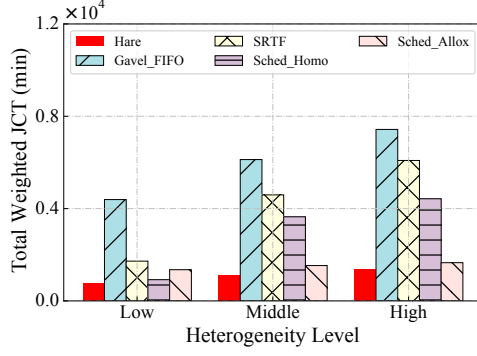


Figure 3.16: Performance under different heterogeneity levels.

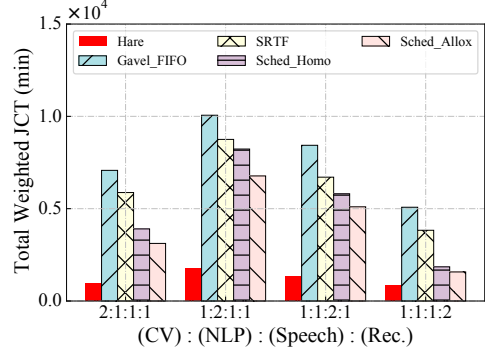


Figure 3.17: Performance under different fractions of jobs.

### 3.7.3 Results of Simulations

Large-scale experiments are conducted using the simulator. As we have shown in Fig. 3.12, the maximum performance gap between the testbed and simulator is only 5%, which demonstrates that the simulator can offer sufficient simulation accuracy. The gap is mainly because the error in prediction of training time and switching cost.

We study the influence of number of GPUs in Fig. 3.14. The number of ML jobs is set to 200. The weighted JCT of all schemes decreases as more GPUs are used. Hare always outperforms other schemes under all cases. Sched\_Allox is slower than Hare by about 2x, but it is still significantly faster than others, thanks to its heterogeneity-aware design. Although Gavel\_FIFO schedules jobs with the consideration of heterogeneity, it still has the largest weighted JCT since it has no optimization in scheduling.

We then consider 160 GPUs and change the number of jobs from 100 to 300 to see how it affects the performance. As shown in Fig. 3.15, as the number of jobs increases, the total weighted JCT grows under all schemes. Meanwhile, the performance gaps between Hare and other schemes become bigger. For example, Hare outperforms others by 54.6%-80.5% when processing 300 jobs. It demonstrates that Hare can use these GPUs in a more efficient way, to minimize the total weighted JCT.

We study the influence of GPU heterogeneity in Fig. 3.16. We consider 160 GPUs and 200 jobs. We set different heterogeneity levels by selecting a different combination of GPUs. For the low heterogeneity level, we only choose V100 GPUs for training. We select the combination of (V100×K80) GPUs as the middle heterogeneity level while selecting the combination of (V100×T4×K80×M60) GPUs as the high heterogeneity level. We find the gaps between Hare and other schemes become bigger as the increasing of heterogeneity level. The main reason is the higher heterogeneity level results in lower resource utilization in heterogeneity-oblivious schemes. Although Sched\_Allox suffers a slight influence from the heterogeneity level, its performance still lags behind Hare by 2× since there is no consideration of intra-job parallelism optimization. We also find that Hare and Sched\_Homo have close performance when there is a low-level heterogeneity, because intra-job parallelism optimization has the dominant influence in such scenarios.

We investigate how job type affects the performance by changing their proportions. The results are shown in Fig. 3.17. In the default setting, each type of jobs account for 25%. In each experiment, we then increase one of them and keep others the same. The x axis of Fig. 3.17 shows the ratio of different job types. When we increase the

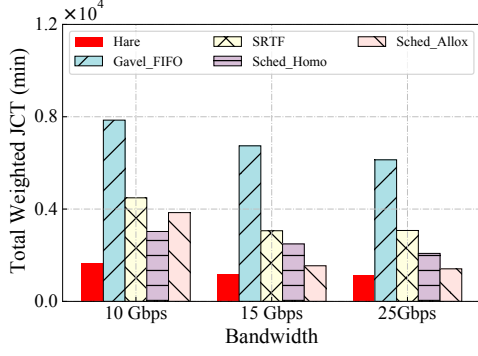


Figure 3.18: Performance under different bandwidth.

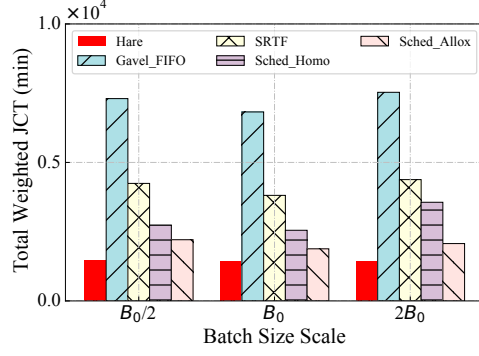


Figure 3.19: Performance under different batch sizes.

proportion of NLP jobs, the total weighted JCT of all schemes increases since NLP jobs involve heavier training workloads (i.e., more training rounds and more training time). On the other hand, all schemes have smaller weighted JCT when more recognition jobs are added, because they have less workloads. Although Hare is affected by the job proportion, it always achieves the best performance due to the sophisticated scheduling algorithm.

We change the speed of the network connecting GPUs and study its influence in Fig. 3.18. The results are in alignment with our intuition that faster networks can accelerate the ML training. However, such acceleration is not linear with the network speed since the training time becomes the main bottleneck as the decreasing of the synchronization time. For example, Hare’s weighted JCT decreases by only 31.2%, even though increase the network speed from 10Gbps to 25Gbps.

Fig. 3.19 shows the performance under different batch sizes, where  $B_0$  stands for the default batch size configuration. We can see that batch size has no big influence to all schemes except Sched\_Homo. That is because larger batch size leads to longer training time, and there is more GPU idle time in Sched\_Homo.

### 3.8 Summary

We present Hare, a system enabling efficient multiple DML job scheduling on heterogeneous GPU cluster. Considering frequent task switching may happen in Hare, we propose fast task switching optimization in Hare to reduce the overhead of task switching. Besides, we propose a heterogeneity-aware task scheduling algorithm to minimize the total weighted job completion time. We demonstrate the performance of Hare through experiments on both small-scale testbed and large-scale trace-driven simulator. Hare can significantly outperform existing works.

## Chapter 4

# Efficient Distributed Graph Learning on Dynamic Graphs

### 4.1 Problem Statement

Graph Neural Network (GNN) has achieved great success in learning graph data in many fields, i.e., drug discovery [124, 125], recommendation systems [126, 127], and social networks [128]. Existing GNN can handle only static graphs, where vertices and edges, as well as associated features, have no change across time. However, many practical applications generate dynamic graphs whose vertices and edges change over time. Typical examples include traffic graphs that describe real-time traffic flows of roads [40, 46, 129], and social networks where edges representing friend connections could be created or removed as the change of social relationship [130, 131]. The strong demand for processing dynamic graphs motivates the design of the Dynamic Graph Neural Network (DGNN). As shown in Figure 4.1, a dynamic graph is divided into a number of snapshots, each of which represents the graph at a specific time. These snapshots are fed to structure encoders (e.g., GCN, respectively, followed by time encoders (e.g., RNN) to exploit the temporal relationship across snapshots. By stacking multiple layers of structure encoders and time encoders, DGNN has a strong capability of capturing spatio-temporal features of dynamic graphs. Based on this basic model, various DGNN variants, EvolveGCN [132] and DySAT [42], have been proposed recently and dynamic graph learning has become a booming research area.

Despite the great research enthusiasm for DGNN model design, its system-level support has been seldom studied. Since dynamic graphs could be very large, DGNN training usually runs on distributed systems consisting of multiple GPUs or other accelerators. To build an efficient distributed DGNN system, one of the most crucial challenges is how to partition the dynamic graph among multiple GPUs to minimize cross-GPU traffic, which has been recognized as the main system bottleneck by existing work [59, 60]. A straightforward idea is to partition the dynamic graph into snapshots and assign them to GPUs, as shown in Figure 4.2(a). This method is referred to as PSS. However, PSS would incur high communication costs when handling dynamic graphs with long temporal information since time encoders need to share temporal embeddings across GPUs.

To eliminate temporal embedding transmissions, the method of PTS has been proposed [59]. As shown in Figure 4.2(b), PTS divides dynamic graphs into temporal sequences. Each sequence contains the same vertex's embeddings of different time,

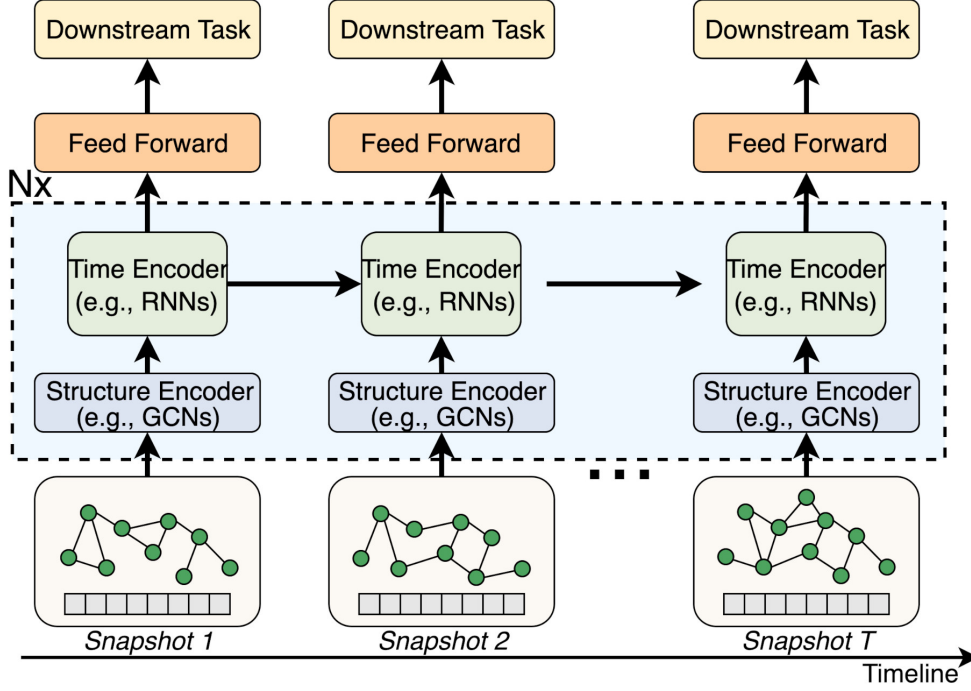


Figure 4.1: Dynamic graph neural network.

and it is the basic assignment unit to GPUs. PTS hides all communication of temporal embedding sharing within each GPU, but pays the cost of aggregating spatial embeddings across GPUs. Recently, Chakaravarthy et al. [60] have proposed a joint partitioning method to take the benefits of both PSS and PTS methods. As illustrated in Figure 4.2(c), it applies PSS to assign snapshots to GPUs and runs structure encoders. Then, generated embeddings are shuffled by PTS, so that the ones of the same temporal sequence are gathered into the same GPU. This method is referred to as PSS-TS. Although both spatial and temporal communication is avoided, the embedding shuffling process incurs additional communication costs.

We have conducted a quantitative study (in §6.2) on the aforementioned methods by comparing their performance on various datasets. Our results indicate that these methods demonstrate different performance on these datasets, and there is no single method that always outperforms the others. We find that main reason of this inconsistency is an implicit assumption of these methods that dynamic graphs are uniform in both spatial and temporal structures. However, many graphs are not uniformly structured, with some snapshots being very dense while others are sparse. Additionally, temporal sequences could have different lengths, with some vertices existing for a long time and being associated with long temporal sequences while others have short sequences. Our experiments in §6.2 have also revealed that such spatio-temporal non-uniformity is prevalent in popular datasets. This important observation motivates us to re-examine the graph partitioning problem for DGNN training, and to design a new dynamic graph partitioning method aware of such spatio-temporal non-uniformity, so that it can always outperform existing ones on a variety of datasets.

As a positive response to this challenging problem, we design DGC, a distributed system for efficient DGNN training, implementing a new method called Partitioning by Graph Chunks (PGC). Different from existing works that treat snapshots or tempo-

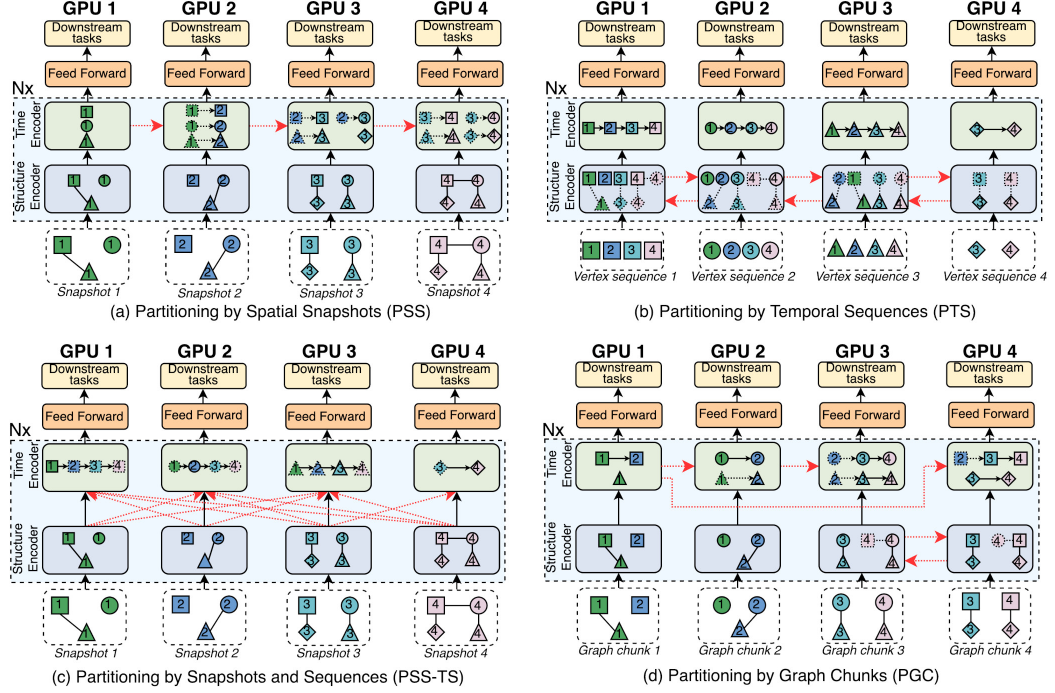


Figure 4.2: Different dynamic graph partitioning methods for distributed training. Different node shapes (such as rectangles, circles, triangles, and rhombuses) represent different vertices, while colors and numbers signify vertices belonging to different snapshots. Red dotted arrows indicate communication between GPUs. Within each GPU, vertices aggregated from other GPUs are represented by dotted lines.

ral sequences as basic partitioning units, we propose to partition dynamic graphs into chunks that are essentially sub-graphs across the spatial and temporal boundaries. As shown in Figure 4.2(d), each graph chunk may contain vertices and edges belonging to different snapshots and temporal sequences. We design a graph chunk generation algorithm based on the graph coarsening technique with a full consideration of spatio-temporal non-uniformity, so that each graph chunk has a modest training workload and few edge connections to other chunks. By a simple heuristic to assign these chunks to GPUs, DGC can achieve better workload balance and reduced communication cost, to significantly improve DGNN training efficiency.

In addition, we propose two techniques to optimize the run-time of DGC by exploiting unique characteristics of graph chunks for further performance improvement. The first one is called chunk fusion. The graph chunks assigned to a GPU need to first go through the structure encoder. A default scheme is to load and train these chunks one by one, which would be inefficient because of redundant data loading and low GPU utilization. To address this issue, we propose to fuse these chunks into larger ones before loading, while considering the GPU memory constraint. Furthermore, the temporal sequences sent to the time encoder could have different lengths. In order to pack them for GPU processing, we need to align these sequences by padding a large number of zeros, which could waste GPU memory. Thus, we propose to fuse these sequences by concatenating short ones to reduce padded zeros. However, an intuitive sequence concatenation scheme would generate incorrect outputs of time encoders, and thus impose negative influence on training accuracy. We design a masking scheme for the time



encoder, so that it can generate correct embeddings while padded zeros can be reduced.

Second, we propose adaptive stale embedding aggregation to further reduce communication cost among GPUs. This is motivated by the observation that vertices may generate similar embeddings in different training epochs (§4.5.2). DGC allows GPUs to reuse stale embeddings from previous epochs if they are sufficiently similar, to reduce data traffic between GPUs. However, using stale embeddings could slow down the training convergence or even decrease the final training accuracy. It is quite challenging to estimate embedding similarity and decide when they can be reused, to balance communication cost and training convergence. We propose an adaptive stale aggregation scheme, which decides whether stale embeddings could be used according to the current training loss.

We deploy DGC on an 8-GPU testbed and conduct experiments using four different dynamic graphs and three representative DGNN models (including T-GCN [40], DySAT [42], and MPNN-LSTM [133]). The experimental results show that DGC achieves a  $1.25\times$  -  $7.52\times$  speedup over the state-of-the-art. We also conduct ablation experiments to study the benefits of our proposed run-time optimization techniques.

## 4.2 Motivation

Attributes	Amazon	Epinion	Movie	Stack
# of snapshots	121	500	289	93
Total # of vertices	103M	72M	43M	83M
Total # of edges	5.7M	13M	27M	47M

Table 4.1: Dynamic Graph Datasets.

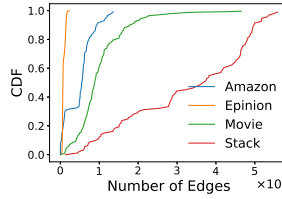


Figure 4.3: CDF of number of spatial neighbors.

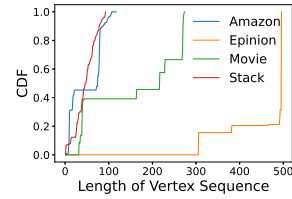


Figure 4.4: CDF of vertex sequence lengths.

### 4.2.1 Spatio-temporal non-uniformity of dynamic graphs

We demonstrate the spatio-temporal non-uniformity of dynamic graphs using four datasets, whose details are shown in Table 4.1. In Figure 4.3, we plot the cumulative distribution function (CDF) curves of the number of edges within snapshots. We observe that some snapshots have very few edges, while others could be very dense, indicating diverse spatial features among snapshots. The CDF about lengths of vertex sequences is shown in Figure 4.4. Some vertices exist for a long time and thus have long temporal sequences, while others are short.

### 4.2.2 Performance of dynamic graph partitioning methods on different datasets

We use the four datasets in Table 4.1 to train DySAT [42] models on 4 NVIDIA V100 GPUs. The average epoch time of PSS, PTS, and PSS-TS is shown in Figure 4.5(a). Although all methods are performing the same training task using the same

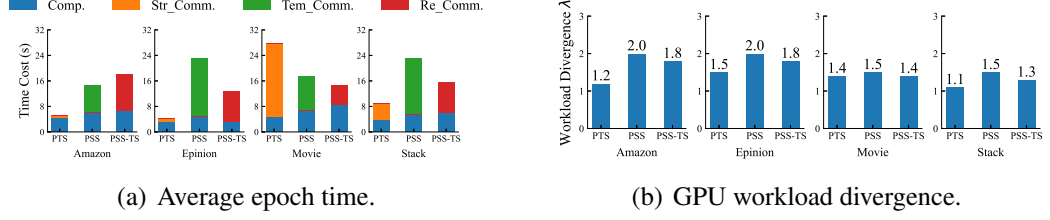


Figure 4.5: Performance of dynamic graph partitioning methods on different datasets.

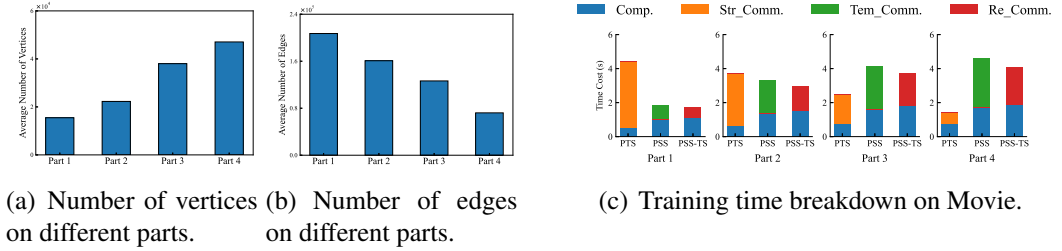


Figure 4.6: Performance of dynamic graph partitioning methods within a single dataset.

dataset and model, they are different in graph partitioning and thus assign different workloads to GPUs. We can see that the PTS has the shortest epoch time on Amazon, Epinion, and Stack datasets, but longer than PSS and PSS-TS methods on the Movie dataset. The breakdown of computation time and communication time is also shown in this figure. For the Epinion dataset, all methods have similar computation time, but PSS has much longer communication time, because more nodes are involved in temporal computation and their embeddings are shared across GPUs. However, since the Movie dataset has dense spatial structures, PTS breaks this structure and incurs higher communication overhead. Although PSS-TS avoids communication overhead within both spatial and temporal computations, it incurs significant overhead because of embedding re-assignment, especially when the number of vertices is large (e.g., Amazon and Stack datasets). The fact in Figure 4.5(a) demonstrates that different datasets show distinct spatio-temporal features and neither method can always win over all datasets.

We also conduct experiments to study GPU load balance of existing methods. For the PSS method, we follow the strategies proposed in [59, 60] to assign the same number of snapshots to each GPU. Similarly, we let each GPU get the same number of sequences in the PTS method. We define a metric  $\lambda = \frac{T_{max}}{T_{min}}$  to evaluate the level of GPU load balance, where  $T_{max}$  and  $T_{min}$  is the maximum and minimum epoch time, respectively, among GPUs. If the value of  $\lambda$  is close to 1, training workloads are well balanced. Otherwise, faster GPUs need to wait for slower ones, leading to low hardware utilization. As shown in Figure 4.5(b), we find that PTS method has a good load balance with  $\lambda = 1.1$  under the Stack dataset, but its load balance becomes worse when training other datasets. PSS and PSS-TS have bigger  $\lambda$ , indicating stronger imbalance of training workloads among GPUs.



### 4.2.3 Performance of dynamic graph partitioning methods within a single dataset

We then dig into the Movie dataset to study its internal spatio-temporal features. We divide the whole dataset into 4 parts by snapshots, and each part has the same number of snapshots. The average number of vertices and edges in each part are shown in Figure 4.6(a) and Figure 4.6(b). We can see that the number of vertices changes significantly across different parts. The fourth part has 3 times more vertices than the first one. Meanwhile, the number of edges also changes, but with a different pattern from vertices. For example, the first part has the most edges but it has only half of vertices of the fourth part.

To study how internal spatio-temporal features affect graph training, we measure the epoch time of three methods on 4 sub-datasets. As shown in Figure 4.6(c), these methods show distinct performance. PSS and PSS-TS outperform PTS when training the first and second sub-datasets, thanks to its much shorter communication time. However, PTS has better performance on the third and fourth sub-datasets. We also find that the communication overhead of PSS and PSS-TS grows as the increasing of the number of vertices. In contrast, PTS has a longer epoch time on sub-datasets with more edges. That is because PSS partitions dynamic graph data by snapshots, and denser snapshots incur more data sharing over networks. PTS conducts data partition by sequences, i.e., cutting edges within snapshots, and thus more edges would generate more traffic.

We also conduct experiments for other 3 datasets and have similar observations. For even a single dataset, its different parts show distinct spatio-temporal features. However, existing works are unaware of such graph internal diversity and apply a single partitioning strategy for the whole dataset.

## 4.3 System Overview

In this section, we present an overview of the DGC design. We first set our design goals as follows.

**High training efficiency.** Due to massive data dependencies (including spatial and temporal dependencies) among vertices in dynamic graphs, distributed DGNN training suffers from high communication costs that would be the performance bottleneck. DGC needs to reduce communication costs to accelerate training process.

**High GPU utilization.** Multiple GPUs are used to train DGNNs for handling large dynamic graphs. GPU utilization is a crucial metric for efficient resource management. DGC should ensure high GPU utilization during DGNN training.

**Consistent training convergence.** While introducing various optimizations to accelerate DGNN training, DGC needs to ensure that these designs do not compromise training convergence, preserving the quality of the final model.

Figure 4.7 illustrates an overview of DGC’s design. In order to handle large dynamic graphs, DGC uses multiple GPUs that collaboratively train the DGNN model. Specifically, DGC maintains some training workers, and each worker is bonded to a GPU. The system workflow is as follows. ❶ First, a PGC (Partitioning by Graph Chunk) module partitions the dynamic graph into multiple graph chunks and assigns them to workers. ❷ Based on the partitioning and assignment results, each GPU worker loads their assigned graph chunks, and then trains the corresponding DGNN model for multiple epochs. As mentioned in §??, a DGNN model could include multiple blocks, and

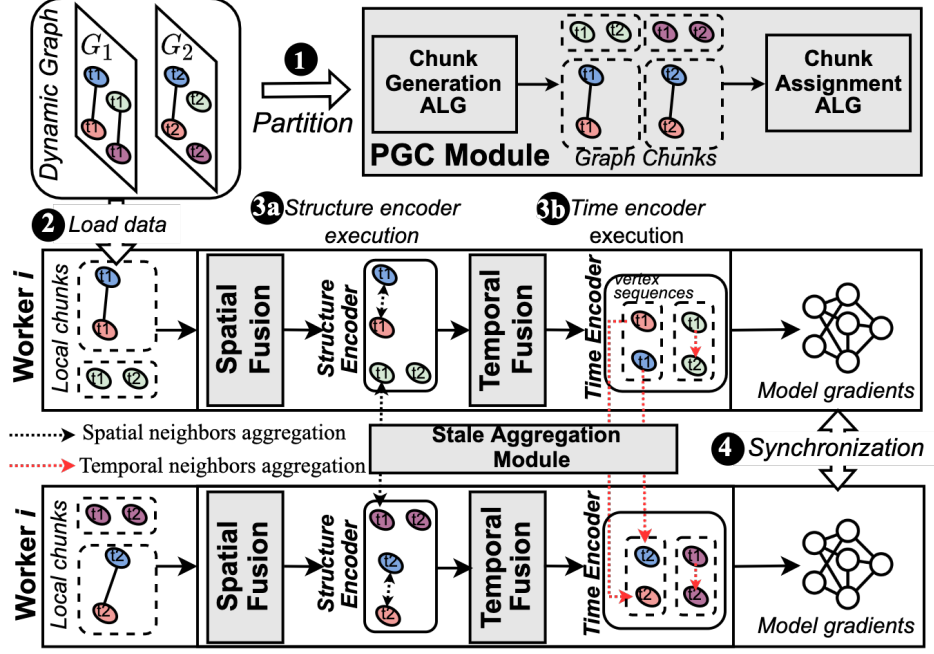


Figure 4.7: System Overview. DGC contains an offline PGC module and an optimized run-time. The PGC module partitions the dynamic graph into chunks and assigns them to GPUs. The run-time contains two new modules of chunk fusion and adaptive stale embedding aggregation.

each block consists of a structure encoder and a time encoder. In Figure 6.3, we show a DGNN model with a single block for simplicity. ③a The structure encoder computes spatial embeddings of local vertices. Note that spatial neighbors may be located on other workers, their embeddings need to be transmitted over the network. ③b Similarly, time encoders of different workers need to share embeddings of temporal neighbors. Since these embeddings could be very large, frequent embedding sharing would incur high communication costs. ④ Finally, each GPU worker calculates gradients based on a loss function and synchronizes them with other workers, so that they can update DGNN model weights and proceed to the next training epoch.

We can see that DGC is different from traditional data parallelism and model parallelism, which are popular distributed training approaches adopted by CNN or transformer models, because of the complicated spatio-temporal dependency. The whole system performance is mainly affected by communication costs among GPUs and their workload balance. In order to achieve our design goals, we design the following three key modules.

**PGC:** The PGC module partitions the dynamic graph, by introducing the concept of graph chunks to minimize cross-GPU communication cost while maintaining workload balance (§4.4).

**Chunk fusion:** This module fuses multiple chunks assigned to each GPU to reduce data loading costs, so that the GPU utilization can be significantly improved (§4.5.1).

**Adaptive stale embedding aggregation:** We observe that some vertex embeddings have no big changes in different training epochs. Thus, we are motivated to propose a stale aggregation module that enables some GPUs to reuse some previously received embeddings if there is only a trivial difference. Many embedding transmissions can be

avoided to reduce communication costs (§4.5.2).

## 4.4 Partitioning by Graph Chunks

The issues of PTS and PSS stem from their high-level semantic graph partitioning, i.e., in units of snapshots or sequences, without considering the potential influence on running efficiency of the distributed system. Since snapshots and temporal sequences could be very large, it leaves little optimization space for the following workload assignment algorithm. No matter how sophisticated assignment algorithms are designed, it is still difficult to achieve good workload balance among GPUs while minimizing cross-GPU traffic.

To fundamentally solve these issues, we propose the method of PGC, by jointly considering graph features and hardware resources. PGC partitions graphs into chunks that are sub-graphs across spatial and temporal boundaries of original dynamic graphs. Each graph chunk has a modest training workload and few edge connections to other chunks, so that even a simple chunk assignment algorithm can achieve significant efficiency improvement.

However, designing an efficient PGC is challenging. Since the weaknesses of PTS and PSS are mainly because of their coarse-grained partition at the snapshot or sequence level, a straightforward improvement is to treat dynamic graphs as a super graph by linking vertices with temporal relationships. We can then partition this super graph into several parts, each of which is assigned to a GPU for training. Such a kind of graph partitioning has been widely studied by existing works [55, 134, 135]. Even though it is an NP-hard problem, there exist methods with good theoretical and empirical results. However, these methods have high computational overhead, which can be hardly applied to dynamic graphs with millions or even billions of vertices and edges. We address this overhead challenge by borrowing the idea of graph coarsening [136–139], and customizing it for dynamic graph partitioning. In addition, we design a fast algorithm to assign chunks to GPUs.

### 4.4.1 Chunk Generation

The chunk generation algorithm is based on label propagation and its basic idea is to assign a unique label to each vertex, which is then propagated along graph edges and be updated iteratively according to a label updating policy. Finally, vertices with the same label can be grouped together to form a chunk.

Two key challenges must be addressed to make this algorithm work efficiently for dynamic graphs. First, traditional label propagation is constrained within snapshots (because there is no edge between snapshots) and temporal features cannot be fully exploited. Therefore, we add virtual temporal edges between temporal vertices so that labels can be propagated across snapshots.

Second, even with these virtual edges, the label propagation algorithm could be difficult to generate chunks with minimum inter connections as we desire, because the algorithm lets labels have the same opportunity to travel along all edges. However, spatial edges and temporal edges have different communication costs. For example, T-GCN involves two GCN layers and one GRU layer for each DGNN block, which means that vertices aggregate their spatial neighborhoods twice, while only aggregating

temporal neighborhoods once. To reflect this unique characteristic, we propose to customize edge weights during label propagation according to their communication cost. Specifically, we initialize the label of each vertex  $v_{i,t}$  as follows:

$$c(v_{i,t}) = \sum_{\tau=1}^{t-1} |V_\tau| + i, \quad (4.1)$$

where  $|V_\tau|$  is the number of vertices in snapshot  $G_\tau$  and  $\tau \in [1, t-1]$ , so that each vertex can get a unique label. After initialization, the algorithm runs several iterations of label propagation. In each iteration, vertices propagate labels to both spatial and temporal neighbors and update their labels in a *weighted* manner. Specifically, each vertex  $v_{i,t}$  receives multiple labels from its neighbors, and these labels are maintained in a set  $\mathcal{L}(v_{i,t})$ . The set of vertices sending the same label  $c$  is denoted by  $S(c)$ . Each label  $c$  is associated with a weight  $weight(c)$  that is the total amount of traffic for embedding sharing from vertices in  $S(c)$  to  $v$ . Note that  $weight(c)$  may vary depending on the specific DGNN model and can be easily obtained through profiling. Vertex  $v_{i,t}$  updates its label by:

$$c(v_{i,t}) = \underset{c \in \mathcal{L}(v_{i,t})}{\text{argmax}} \ weight(c), \quad (4.2)$$

which chooses the label with the maximum weight. The rationale is as follows. Recall that our final goal is to create graph chunks with minimum inter connections. Since directly minimizing inter-chunk connections could be difficult, we convert the problem into an equivalent one of maximizing the communication cost within chunks. The equivalence can be proved by formulating both problems and showing that the sum of their objective functions is a constant, i.e., the total cost of all edges. Therefore, we choose a neighboring label with the most weight, so that they can be grouped together as a chunk. The above process is repeated until convergence, i.e., no labels can be changed. Note that we control the maximum size of chunks by constraining the propagation of some labels if they are attached to too many vertices.

**Discussion.** To maximize GPU utilization, an alternative method is to let graph chunks expand until they reach GPU memory capacity during chunk generation. However, due to the convergence of label propagation, this method cannot guarantee that each generated chunk perfectly saturates GPU memory. Imposing chunks to expand to GPU memory would falsely group vertices, leading to high cross-GPU traffic. Our design respects the convergence of label propagation and uses a fast algorithm to fuse chunks (in §4.5.1) for high utilization of GPU memory.

#### 4.4.2 Chunk Assignment

After chunk generation, we need to assign generated chunks to GPUs, by considering cross-GPU communication cost and workload balance among GPUs. An important step in chunk assignment algorithm design is to evaluate chunk workload. A simple and straightforward method of evaluating chunk workload is to count the numbers of vertices and edges [39]. However, this method cannot provide sufficient estimation accuracy because the execution time of a chunk on a GPU is determined by many factors, such as the number of vertices, number of edges, sequence lengths, feature dimension, and others. Our experiments have also confirmed this point. As shown in Figure 4.8,

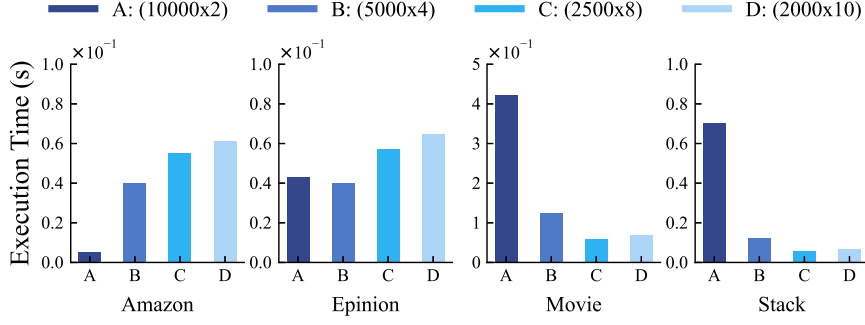


Figure 4.8: The execution time of different chunks with the same number of vertices.

---

**Algorithm 2** Chunk Assignment Algorithm
 

---

**Input:** A set of graph chunks  $A$ , and a set of GPUs  $M$

**Output:** Assignment decisions:  $x_a, \forall a \in A$ , denoting the GPU of chunk  $a$ ;

0: A set of chunks already assigned to GPU  $m$ :  $Q_m \leftarrow \emptyset, \forall m \in M$ ;

0: Profile workloads of chunks with MLPs, denoted by  $g_a$ ;

0: Sort chunks in decreasing order of  $g_a$ , as  $\tilde{A}$ ;

0: **for**  $a \in \tilde{A}$  **do**

0:     **for**  $m \in M$  **do**

0:

$$s_m = (\bar{g} - \sum_{a' \in Q_m} g_{a'}) \cdot \sum_{a' \in Q_m} h(a, a'); \quad (4.3)$$

0:     **end for**

0:      $m^* = \arg \max_m s_m$ ;

0:      $x_a = m^*, Q_{m^*}.append(a)$ ;

0: **end for**

---

we generate four chunks with the same number of vertices, but their execution time is different and the maximum gap could be 8 times.

To address this challenge, we propose a learning-based method to accurately evaluate the workload of each chunk. Specifically, we use multi-layer perceptions (MLPs) to predict the execution time of training operations associated with a chunk as its workload. Furthermore, we find that there are two kinds of operations, e.g., spatial ones and temporal ones, which consume different time. Therefore, we train two separate MLPs to predict the execution time of structure and time encoders, respectively. More details about the implementation details of prediction MLPs are given in §4.6.

We design a heuristic algorithm to assign generated chunks to GPUs, whose pseudocodes are shown in Algorithm 2. We first predict the workload of each chunk  $a$  using the proposed MLPs. Chunks are sorted in decreasing order of their predicted workloads. Then, for each chunk  $a$ , we compute assignment scores, which are defined in Eq. (4.3), for all available GPUs. The score consists of two parts. The first part  $(\bar{g} - \sum_{a' \in Q_m} g_{a'})$  indicates the workload balance among GPUs, where  $\bar{g}$  is the average workload and  $Q_m$  is the set of chunks assigned to GPU  $m$ . The second part  $\sum_{a' \in Q_m} h(a, a')$  is the communication cost between the chunk  $a$  and the ones already assigned to GPU  $m$ .

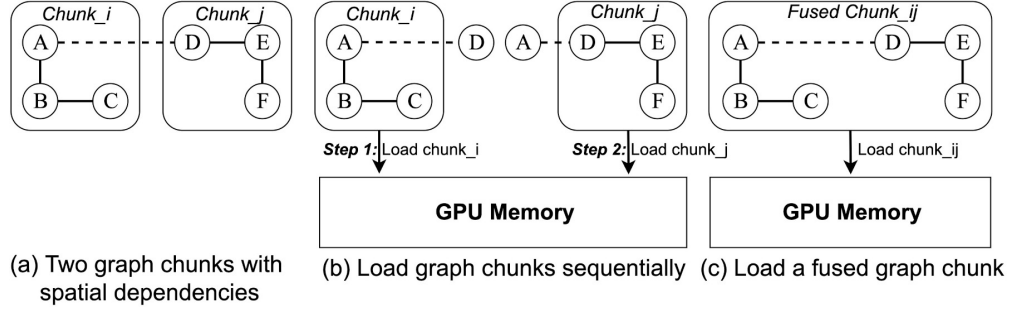


Figure 4.9: An illustration of spatial fusion.

## 4.5 Run-time Optimization

After chunk assignment, the whole training system is ready for running. In this section, we propose two techniques to optimize DGC run-time to accelerate the training process.

### 4.5.1 Chunk Fusion

Each GPU is assigned by a number of graph chunks. A default approach is to load and process these chunks one by one, which would lead to substantial redundant data loading and low GPU utilization.

#### Spatial Fusion

We use an example in Figure 4.9 to show the motivation of spatial fusion. Two graph chunks with cross-chunk spatial dependency (e.g., the edge between  $A$  and  $D$ ) need to be loaded into the GPU for training. In a default scheme, when loading each chunk, we need to load not only the vertices within this chunk but also the ones of other chunks. For example, in Figure 4.9(b), when processing chunk<sub>i</sub>, we must load vertex  $D$  from chunk<sub>j</sub> because vertex  $A$  requires information from  $D$ . Similarly, when processing chunk<sub>j</sub>, we also need to load vertex  $A$ . Consequently, vertices  $A$  and  $D$  are loaded twice.

DGC introduces spatial fusion to reduce data loading costs and improve GPU utilization. By fusing multiple chunks together, we can load them simultaneously and only load the vertices with cross-chunk dependency once, as illustrated in Figure 4.9(c). Moreover, fused chunks can be executed together to fully utilize GPU resources.

Although spatial fusion can significantly increase GPU utilization, a GPU could be assigned with a large number of graph chunks and fusing all chunks may exceed the GPU memory limit. Therefore, we propose a simple yet effective heuristic algorithm to select a subset of graph chunks for fusion with respect to the GPU memory constraint. Specifically, we estimate the potential redundant data loading, in terms of the amount of data transmission, among chunks and then iteratively fuse two chunks with the maximum data transmission. When the size of any fused chunk is close to GPU memory limit, we stop to fuse them anymore.

To estimate the GPU memory consumption of a chunk, we exploit the observation that memory consumption remains relatively consistent across training epochs for each chunk. As a result, it is sufficient to determine the GPU memory consumption through a

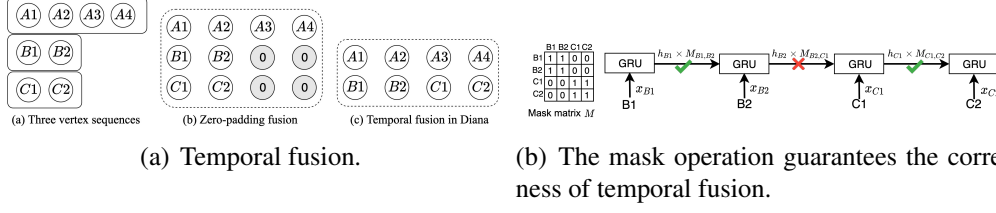


Figure 4.10: (a) An illustration of temporal fusion. (b) We take a GRU layer as an example, which is adopted in the time encoder of T-GCN. The sequence is  $(B1, B2, C1, C2)$ , given in Figure 4.10(a)(c).

single execution. Specifically, right after completing the first training epoch, we monitor the memory usage during the training process to assess the memory consumption of each chunk.

### Temporal Fusion

By carefully examining the input of time encoders, we find that vertex sequences have varying lengths, as an example shown in Figure 4.10(a)(a). A common practice is to pad zeros, so that they can be packed together and processed by GPUs, as shown in Figure 4.10(a)(b). However, padding zeros incurs redundant computation costs and wastes GPU memory. To further increase GPU utilization, we propose to fuse embeddings by concatenating shorter sequences instead of padding zeros. As illustrated in Figure 4.10(a)(c), we concatenate sequences  $(B1, B2)$  with  $(C1, C2)$  to form a new sequence of length 4, enabling simultaneous processing with another sequence  $(A1, A2, A3, A4)$ .

Although this method can improve GPU utilization by avoiding zero padding, the time encoder may generate incorrect output due to unnecessary message passing between vertices belonging to different sequences. For instance, if we concatenate  $(B1, B2, C1, C2)$  as one sequence, vertex  $C1$  receives an unwarranted hidden state from  $B2$ , resulting in wrong outputs. To address this issue, DGC uses a mask to ensure correct output, as shown in Figure 4.10(b). Taking the sequence  $(B1, B2, C1, C2)$  as an example, we calculate the update of  $C1$  in time encoder of T-GCN as follows:

$$u_{C1} = \sigma(W_u h_{B2} M_{B2,C1} + W_u x_{C1} + b_u), \quad (4.4)$$

$$M_{B2,C1} = \begin{cases} 1, & \text{if } B2, C1 \text{ belong to a sequence,} \\ 0, & \text{otherwise,} \end{cases} \quad (4.5)$$

where  $W_u$  and  $b_u$  are learnable weights and bias in the update gate, respectively. The term  $h_{B2}$  represents the hidden state of  $B2$ , and  $\sigma$  is the activation function. We use the mask  $M_{B2,C1}$  to prevent the hidden state of  $B2$  from being added to the update gate output of  $C1$ .

### 4.5.2 Adaptive Stale Embedding Aggregation

To further improve system efficiency by reducing network traffic, we propose adaptive stale embedding aggregation by exploiting the embedding similarity. This idea is motivated by an important observation that some vertices generate similar embeddings in different epochs. We collect all embeddings in some epochs when training DySAT

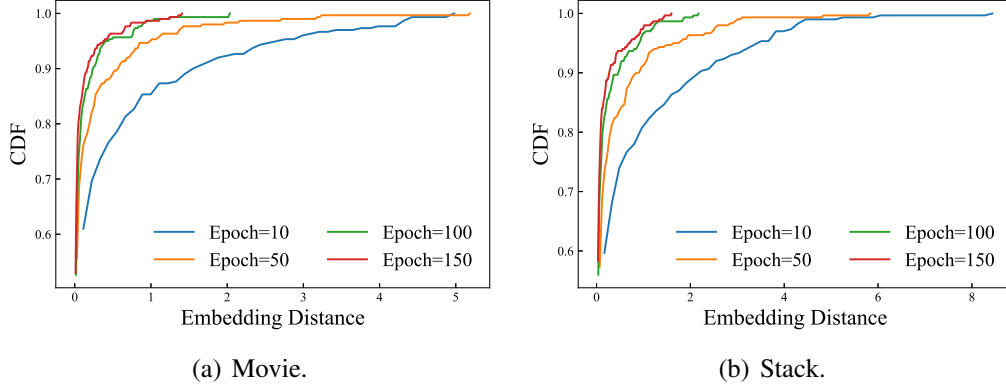


Figure 4.11: CDF of L2 distances between embeddings.

models on the Movie and Stack datasets. The CDF of L2 distances between embeddings in a randomly selected epoch and those in the previous epoch is shown in Figure 4.11. A smaller distance implies that embeddings have slight changes. We find that about 85% embeddings in the 10-th epoch have distances less than 1 on the Movie dataset, while the corresponding percentage of the Stack dataset is 78%. Moreover, we find that embedding distances become smaller as training proceeds. For example, over 95% embeddings of the Movie dataset have distances less than 0.3 after 100 training epochs.

The above observations motivate us to design a stale embedding aggregation mechanism. Specifically, we compare the current embeddings with the last transmitted ones, and transmit them only when their differences are sufficiently big. To calculate distances, it is only necessary to cache one copy of the embeddings for each vertex, specifically the last transmitted ones. Thus, the memory cost is affordable. Moreover, distances are calculated by the CPU, which has sufficient memory to accommodate these data. An alternative design is to compare the current embedding with the one in the preceding epoch, which would update the cached embeddings in every epoch. However, this design would accumulate embedding errors. Suppose consecutive embeddings have small distances, but accumulative distances between the first one and the final one would be big. This alternative design would never transmit embeddings, and may compromise training convergence. In contrast, our proposed method can well handle such accumulative embedding errors.

Although stale aggregation can effectively reduce network traffic, it faces a critical challenge about how to decide the level of “similarity” for embedding reuse. If strict similarity requirements (e.g., an extreme case is that embeddings should be the exact same) are applied, many embeddings need to be transmitted over the network, and thus little traffic can be reduced. On the other hand, loose similarity requirements may decrease training accuracy. This trade-off is demonstrated by the experimental results shown in Table 4.2, where we let  $D$  denote the maximum L2 embeddings distance in the current epoch and  $\theta$  is a threshold of deciding whether embeddings can be reused. When we increase the value of  $\theta$ , the accuracy of all datasets decreases while more network traffic can be saved. In addition, we find that  $\theta$  has different influences on different datasets. For example, when we set  $\theta = 0.3D$ , about 85.5% network traffic can be saved for Stack, with 0.076 drop in accuracy. To achieve a similar trade-off, we need to set  $\theta = 0.5D$  for the Movie dataset. The above fact demonstrates that a fixed value of  $\theta$  cannot work well for all datasets, and we need to adaptively set it according



Dataset	Metrics	Static stale threshold					
		$\theta = 0$	$\theta = 0.1D$	$\theta = 0.3D$	$\theta = 0.5D$	$\theta = 0.7D$	$\theta = 0.9D$
Amazon	Accuracy	0.690	0.688	0.680	0.677	0.658	0.644
	Reduce comm.	-	0.65%	54.14%	74.22%	87.87%	97.49%
Epinion	Accuracy	0.735	0.699	0.679	0.674	0.657	0.641
	Reduce comm.	-	6.27%	50.32%	75.15%	90.23%	98.38%
Movie	Accuracy	0.824	0.807	0.779	0.742	0.702	0.704
	Reduce comm.	-	44.64%	50.34%	80.80%	92.64%	97.67%
Stack	Accuracy	0.697	0.672	0.621	0.605	0.588	0.591
	Reduce comm.	-	51.36%	85.48%	87.56%	97.22%	98.98%

Table 4.2: Test accuracy with different threshold  $\theta$ .

to dataset characteristics.

Similar ideas of stale aggregation have been also adopted by [18, 140]. However, they define a maximum number of stale epochs that can be tolerated, instead of measuring the embedding similarity. This simple and straightforward method would decrease training accuracy. A recent work, SANCUS [141], measures embedding similarity and defines a static handcrafted threshold to decide whether embeddings can be reused.

We propose an adaptive stale embedding aggregation scheme to reduce communication costs while guaranteeing training convergence. Specifically, for each epoch  $r$ , we define a threshold  $\theta_r$  of embedding similarity to determine whether embeddings could be reused. Its value is calculated by

$$\theta_r = \frac{1}{1 + \exp(\text{norm}(l_{r-1}))} D_r, \quad (4.6)$$

$$\text{norm}(l_{r-1}) = \frac{l_1 - l_{r-1}}{l_1}, \quad (4.7)$$

where  $l_{r-1}$  is the loss value of the epoch  $r-1$  and  $D_r$  is the maximum L2 distance among embeddings in the epoch  $r$ . Note that  $l_1$  is the initial loss value when the training starts and the term  $\text{norm}(l_{r-1}) = (l_1 - l_{r-1})/l_1$  represents the normalized loss decrease in the epoch  $r$ . We use the scaled sigmoid function to adjust the threshold  $\theta$ . The rationale is as follows. In the early stages of training, the model is unstable, and we adopt a small  $\theta$  to ensure most aggregated embeddings are fresh for quick training convergence. As the training progresses, the model tends to be stable, and we increase  $\theta$  to reduce communication costs with a trivial negative influence on training convergence.

**Discussion.** The design objective of the adaptive stale embedding aggregation technique is to strike a balance between network traffic and accuracy. Our experiments demonstrate that the communication costs are significantly reduced while there is a slight decrease in accuracy. Additionally, if there is a strict requirement for accuracy, DGC provides the option to disable the adaptive stale embedding aggregation, thereby maintaining the same accuracy as traditional distributed training systems.

## 4.6 Implementation

**Dynamic Graphs and DGNN Models:** DGC is built on the top of PyTorch [142] and PyTorch Geometric (PyG) [36], which are widely used open-source frameworks

for graph learning. In DGC, we represent dynamic graphs with `DynamicGraphSignal`, an iterator that divides the dynamic graph into multiple snapshots. Each snapshot is deployed with `data.Data`, defined in PyG. The DGNN models used in DGC are implemented with PyG and PyTorch APIs. Specifically, the GNN operations in the structure encoder (e.g., GCN and GAT) are implemented with PyG’s APIs, such as `nn.conv.GCNConv` and `nn.conv.GATConv`. For RNN operations of the time encoder (e.g., GRU and LSTM), we implement them using `torch.nn.GRU` and `torch.nn.LSTM` provided by PyTorch.

**MLP Predictors:** we evaluate chunk workloads by two trained MLPs (§4.4.2). Each MLP consists of an input layer, three hidden layers, and an output layer. We use 256 units in each hidden layer and a ReLU activation function after each layer. The output of the final layer is a single real number, which is the predicted execution time. Both MLPs have three inputs: (1) chunk information, i.e., number of vertices and number of edges; (2) feature information, i.e., vertex feature dimensions; (3) encoder information, i.e., layer dimensions; In this work, we focus on the homogeneous GPU setting. Thus, we do not add the GPU information to the MLP input. We randomly generate 50k chunks offline from four dynamic graphs (Table 4.1) and feed them into the structure and time encoder to measure their execution time as training labels. We adopt a mean absolute percentage error as the loss function and optimize MLPs with an Adam optimizer [143]. Each MLP is trained for 100 epochs.

**Caching Module for Stale Aggregation:** we maintain KVStore servers and clients in GPU workers to cache vertex embeddings for remote aggregation. In our implementation, we deploy one KVStore server. Each GPU worker maintains a KVStore client and communicates with the KVStore server through `torch.distributed.rpc` APIs. Each GPU worker calls a `push()` API to send local embeddings to the KVStore server and updates the caching content. Meanwhile, the GPU worker can call a `pull()` API to aggregate remote embeddings from the KVStore server, and update contents cached in the KVStore client.

## 4.7 Evaluation

### 4.7.1 Experiment Setup

**Environment settings & Metrics.** We deploy DGC on a testbed consisting of eight NVIDIA Tesla V100 GPUs. We use Ubuntu 18.04 with Linux kernel version 5.4, NVIDIA driver 418.21, CUDA 10.1, and cuDNN 8.0.4. The versions of PyG and PyTorch are 2.0.4 and 1.11, respectively. In the overall performance comparison, we train the DGNN model for 100 epochs and measure average training time, including data loading, chunk fusion, remote communication, and GPU computation. We do not include graph partitioning in the training time measurement since it can be executed offline before DGNN training.

**Models & Datasets.** We use the four datasets in Table. 4.1. We divide these four datasets into snapshots with different window sizes, as done by [42]. Specifically, Amazon contains graph data of 3650 days and we let the data of every 30 days be a snapshot. We set window sizes of Epinion, Movie, and Stack datasets as 1, 30, and 10, respectively. For all datasets, we use the in-degree and out-degree as the vertex features, similar to [60, 133]. We choose three representative DGNN models, whose details are as follows, to evaluate the performance of DGC.

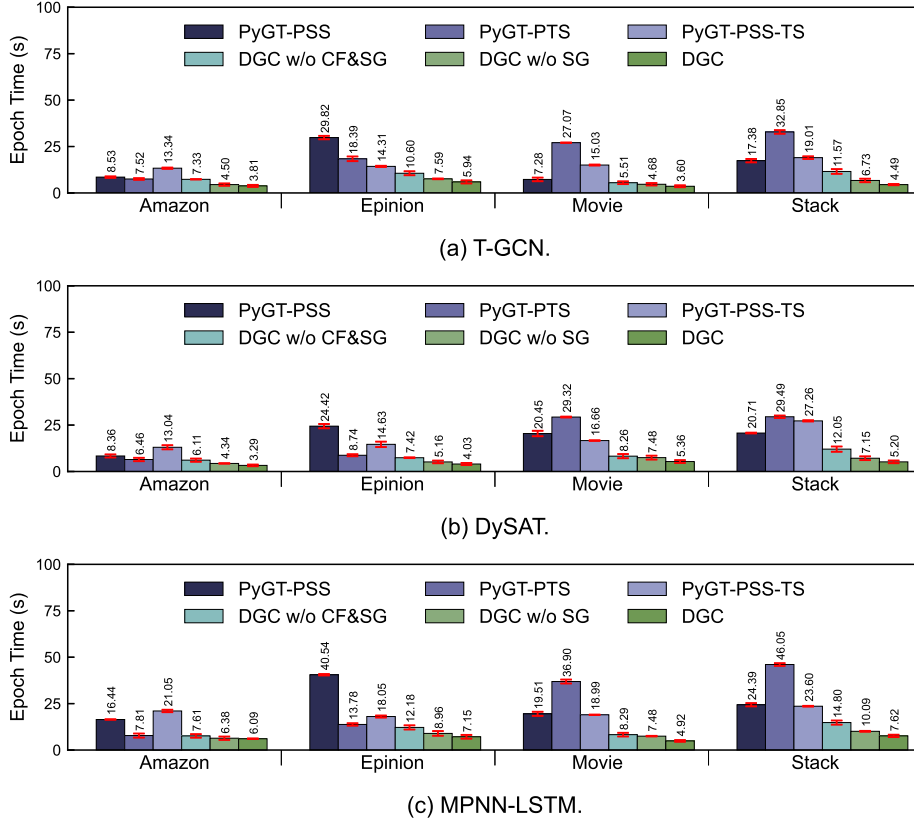


Figure 4.12: Epoch time of different methods.

- **T-GCN** [40]: it utilizes three 2-layer GCN [21] as the structure encoder, and a 1-layer GRU [41] model as the temporal encoder.
- **DySAT** [42]: it uses a 1-layer graph attention network (GAT) [43] and a 1-layer scaled dot-product attention model [44] within each of its DGNN blocks.
- **MPNN-LSTM** [133]: it employs a 2-layer GCN and a 2-layer LSTM [144] as the structure and time encoders, respectively. The structural embedding generation is similar to T-GCN. However, MPNN-LSTM uses a concatenation of outputs from each GCN layer as the input to the time encoder.

**Baselines.** We implement 3 baseline systems with different graph partitioning methods (e.g., PSS, PTS, and PSS-TS), based on the state-of-the-art DGNN framework PyTorch Geometric Temporal (PyGT) [47]. They are referred to as PyGT-PSS, PyGT-PTS, and PyGT-PSS-TS, respectively.

## 4.7.2 Overall Performance Comparison

Figure 4.12 shows the overall speedup over PyGT when using different models. DGC outperforms all baselines by  $1.25\times$  -  $7.52\times$  (on average  $3.95\times$ ,  $3.97\times$ , and  $3.77\times$  for T-GCN, DySAT, and MPNN-LSTM, respectively). Different baselines exhibit varying performance on these datasets. Specifically, on dynamic graphs with fewer spatial dependencies (e.g., Amazon, with only 5.7M edges in total), PyGT-PTS, which breaks

spatial dependency, performs better than PyGT-PSS and PyGT-PSS-TS. This is because fewer spatial dependencies lead to lower communication costs for PyGT-PTS. PyGT-PSS has the worst performance on the Epinion dataset. The reason is that the Epinion dataset has more snapshots, resulting in more temporal dependency. Therefore, PyGT-PSS incurs higher communication costs and longer epoch time, as it neglects the temporal features of dynamic graphs. PyGT-PSS-TS avoids both spatial and temporal communication costs, but adds a shuffling cost to reassign embeddings to GPUs, which depends on the number of vertices. Therefore, for dynamic graphs with a large number of vertices (e.g., Amazon, with 103M vertices in total), it shows worse performance than the other methods. In contrast, DGC partitions dynamic graphs by chunks, considering both spatial and temporal features. It consistently outperforms other approaches and achieves the highest performance. In the following, we give details about results under different DGNN models.

**T-GCN.** Since T-GCN has two GCN layers and one GRU layer in each block, it involves more spatial communication than temporal communication. DGC can exploit this characteristic to obtain more acceleration by reducing spatial communication costs. For instance, DGC achieves a  $7.52\times$  speedup compared to PyGT-PTS on the Movie dataset, where PyGT-PTS ignores the spatial features of dynamic graphs.

**DySAT.** In contrast to T-GCN, DySAT includes only one GAT layer and one temporal attention layer in the DGNN block. However, the unique self-attention mechanism in the temporal attention layer aggregates more temporal neighbors, compared to a GRU or LSTM layer. Specifically, in a GRU layer, each vertex only needs to aggregate the embeddings of its counterpart in the previous snapshot, while the counterparts in all snapshots should be aggregated by a temporal attention layer. The increased number of temporal neighbors results in higher temporal communication costs. Even though, DGC achieves a high speedup of  $6.06\times$  when training a DySAT on the Epinion dataset, compared to PyGT-PSS.

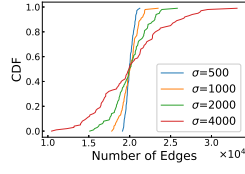
**MPNN-LSTM.** Because of two GCN layers and two LSTM layers in each block, MPNN-LSTM incurs higher communication costs than other models when training on the same dataset. DGC can reduce the epoch time by  $7.5\times$  compared to PyGT-PTS on the Movie dataset while achieving a speedup of  $5.67\times$  over PyGT-PSS on the Epinion dataset. However, the gap between PyGT-PSS-TS and DGC narrows when training MPNN-LSTM. The reason is that DGC’s communication cost increases due to more layers adopted by MPNN-LSTM, but PyGT-PSS-TS’s shuffling cost has almost no change.

### 4.7.3 Ablation Study

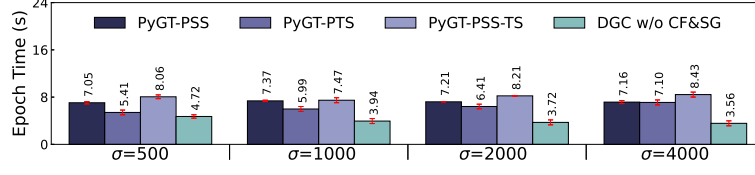
#### Impact of PGC module

As shown in Figure 4.12, DGC with only PGC module, denoted by the bar of “DGC w/o CF&SG”, can still accelerate the training process by  $1.03\times$  to  $4.92\times$ , compared to other methods. PyGT-PTS achieves similar performance with “DGC w/o CF&SG” on Amazon dataset, because this dataset has very few spatial edges within each snapshot and the PTS can generate few cross-GPU traffic, leading to short epoch time.

We further study how data non-uniformity affects the performance of PGC module on synthetic dynamic graphs. The synthetic graphs are generated by setting the total number of vertices, edges, and snapshots to 5M, 2M, and 100, respectively. In order to

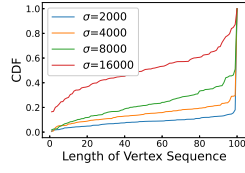


(a) Non-uniformity in spatial features.

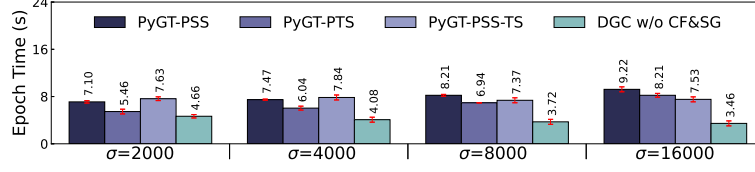


(b) Impact of spatial non-uniformity.

Figure 4.13: Synthetic dynamic graphs with spatial non-uniformity.



(a) Non-uniformity in temporal features.



(b) Impact of temporal non-uniformity.

Figure 4.14: Synthetic dynamic graphs with temporal non-uniformity.

adjust non-uniformity levels of spatial features, we adjust the number of edges for each snapshot according to a normal distribution with a fixed mean value (i.e., 20K), and variable variances  $\delta$ , as shown in Figure 4.13(a). In addition, we change the number of vertices in each snapshot to generate sequences of different lengths, so that we can study the influence of different non-uniformity levels in temporal features, as shown in Figure 4.14(a).

As shown in Figures 4.13(b) and 4.14(b), in both cases, DGC with the PGC module always has the shortest epoch time and it decreases as the levels of spatial and temporal non-uniformity increase. This can be attributed to the following reasons. The PGC module effectively reduces communication costs by aggregating vertices with important spatial features (i.e., those with more spatial dependencies but shorter vertex lengths) or significant temporal features (i.e., those with fewer spatial dependencies but longer sequence lengths). As the non-uniformity levels increase, spatial and temporal features become more obvious, which can be well handled by the PGC module.

### Impact of chunk fusion module

As shown in Figure 4.12, when chunk fusion (CF) is enabled, the epoch time can be further reduced by  $1.39\times$ . In particular, the chunk fusion module is most effective on the Stack dataset. That is because this dataset is large and dense, resulting in large data loading cost, which can be effectively reduced by the chunk fusion module.

To clearly show this benefit, we measure the data loading time per epoch when training MPNN-LSTM on four datasets with and without chunk fusion, and show results in Figure 4.15(a). We can see that chunk fusion can significantly reduce data loading cost by 93.72%, 94.74%, 60.58% and 97.18% on Amazon, Epinion, Movie and Stack, respectively. Without chunk fusion, Stack dataset has the highest data loading cost because it has the largest dynamic graph size. Our chunk fusion searches chunks with the most data dependency to reduce redundant data loading. Moreover, we observe that

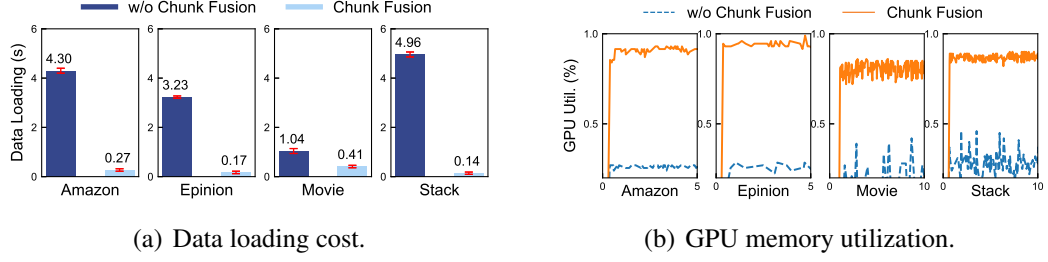


Figure 4.15: Chunk fusion performance.

the improvement on Movie dataset is smaller than that on other datasets. The reason is that vertex degrees of Movie dataset exhibit a significant power-law distribution, which implies that vertices with high degrees would be grouped to graph chunks with large sizes in chunk generation. Thus, generated chunks in Movie dataset have various sizes. The large chunks have few opportunities to be fused with others, leaving a small optimization space in chunk fusion. Thus, compared to other datasets, Movie gets limited improvement by chunk fusion.

Besides reducing data loading costs, chunk fusion lets multiple chunks be processed simultaneously to improve GPU utilization. Figure 4.15(b) shows the GPU utilization when training the MPNN-LSTM model with and without chunk fusion. As a result, chunk fusion significantly improves GPU utilization by 20% to 95%.

### Impact of adaptive stale aggregation module

Dataset	Model	w/o stale aggre.	with stale aggre.							
		Acc	$\theta = 0.3D$		$\theta = 0.5D$		$\theta = 0.7D$		Adaptive threshold	
			Acc	Comm.	Acc	Comm.	Acc	Comm.	Acc	Comm.
Amazon	T-GCN	0.667	0.653	40.39%	0.652	48.74%	0.639	85.97%	<b>0.668</b>	79.32%
	DySAT	0.685	<b>0.687</b>	46.40%	0.657	55.63%	0.636	77.05%	0.656	73.03%
	MPNN-LSTM	<b>0.674</b>	0.653	53.03%	0.648	37.96%	0.631	86.47%	0.654	79.31%
Epinion	T-GCN	<b>0.732</b>	0.703	46.4%	0.691	50.17%	0.651	97.31%	0.701	81.44%
	DySAT	<b>0.738</b>	0.674	63.49%	0.660	69.71%	0.648	93.34%	0.702	79.17%
	MPNN-LSTM	<b>0.661</b>	0.630	68.98%	0.635	69.29%	0.607	97.47%	0.630	94.11%
Movie	T-GCN	<b>0.839</b>	0.828	56.39%	0.825	57.84%	0.774	96.07%	0.830	77.68%
	DySAT	<b>0.829</b>	0.819	62.81%	0.812	70.98%	0.781	93.36%	0.819	78.87%
	MPNN-LSTM	<b>0.727</b>	0.723	31.94%	0.721	32.96%	0.623	83.10%	0.723	53.08%
Stack	T-GCN	0.698	<b>0.702</b>	45.56%	0.696	55.36%	0.599	97.91%	0.694	86.23%
	DySAT	<b>0.703</b>	0.700	68.45%	0.691	79.68%	0.602	95.78%	0.699	89.02%
	MPNN-LSTM	<b>0.654</b>	0.654	67.95%	0.651	68.14%	0.553	97.70%	0.644	91.76%

Table 4.3: Impact of adaptive stale embedding aggregation.

We finally enable the adaptive stale aggregation module. As shown in Figure 4.12, we find that the training time shows even more improvement, by  $1.32\times$  faster. This module dramatically cuts down the volume of communication, thereby expediting the training.

To better understand the benefits of the adaptive stale aggregation module, we further conduct an experimental comparison with static stale thresholds. We have three different settings for static thresholds:  $\theta = 0.3D$ ,  $0.5D$ , and  $0.7D$ . The test accuracy and reduced communication cost when training T-GCN, DySAT, and MPNN-LSTM on four datasets are shown in Table 4.3. In addition, we also show the results of a DGC

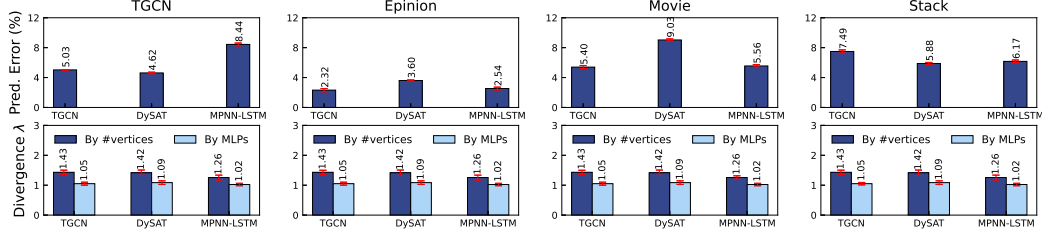


Figure 4.16: Chunk workload prediction error and workload divergence.

variant without stale aggregation, so that we can clearly show the effectiveness of our stale aggregation.

The stale embedding aggregation can reduce communication costs by 32.96% to 97.70% compared to the DGC without stale embedding aggregation. Recall that the design objective of the adaptive stale embedding aggregation is to strike a balance between network traffic and accuracy. According to Table 4.3, the average accuracy drop of our proposed method is 1.56%, but it saves 80.26% cross-GPU traffic, which can significantly accelerate the training speed. We believe this design is attractive to users who care about the time-to-accuracy metric. Moreover, since adaptive stale aggregation is a pluggable module, we can disable it for users who strongly care about accuracy. When training a DySAT model on the Stack dataset, the stale embedding aggregation with a static threshold of  $0.5D$  can decrease communication costs by 79.68%. However, the benefit of traffic reduction is only 32.96% when training the MPNN-LSTM model. Furthermore, a static stale threshold is inadequate in maintaining training convergence. In the Epinion dataset, with a static threshold ( $\theta = 0.5D$ ), the communication cost is reduced by 69.71%. However, the test accuracy decreases from 0.738 to 0.66.

#### 4.7.4 Chunk workload prediction.

Since chunk workload estimation greatly affects the workload balance in chunk assignment, we evaluate the accuracy of our proposed MLPs that predict the execution time of chunks (§4.4.2). We randomly choose graph chunks from four datasets and compare their measured execution time and predicted one by MLPs, which are denoted by  $\text{measured}_a$  and  $\text{predicted}_a$ , respectively. We define the prediction error as:

$$\text{error} = \frac{1}{n} \sum_{a=1}^n \frac{|\text{predicted}_a - \text{measured}_a|}{\text{measured}_a}. \quad (4.8)$$

We set  $n = 1000$  in our experiments. As shown in Figure 4.16, the prediction error is less than 10%, which demonstrates the proposed MLPs have sufficient accuracy to estimate chunk workloads.

We further study the impact of workload prediction on chunk assignment. As shown in Figure 4.16, we report the workload divergence (defined in §4.2.2) of two workload prediction methods. The first one is the baseline method, which estimates the workloads of graph chunks by counting the number of vertices [39, 55]. The second method is to evaluate workloads by MLPs, which is adopted by DGC. The results show that trained MLPs can achieve better workload balance. Specifically, the average workload divergence is about 1.23 when using MLPs, while the divergence increases to 1.67 when we

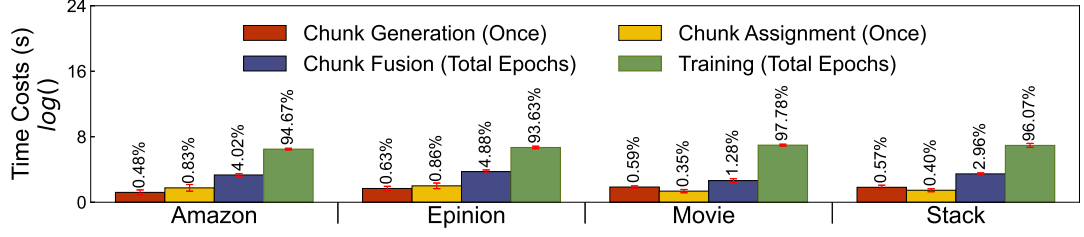


Figure 4.17: Extra overhead introduced by DGC. The numbers above bars are the percentages of total training time.

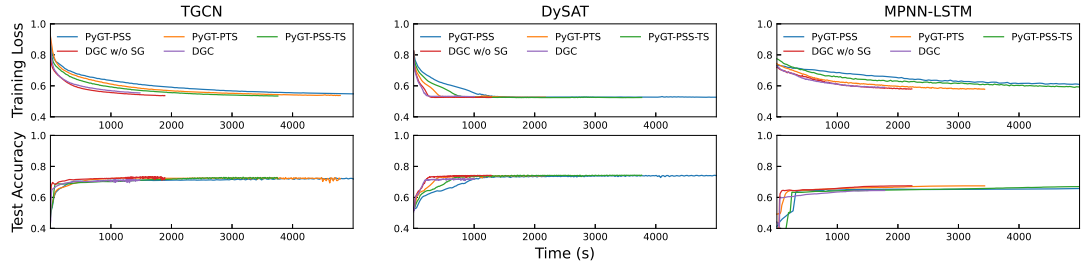


Figure 4.18: Training loss and test accuracy under different methods for three models on the Epinion dataset.

use the number of vertices as chunk workloads.

### 4.7.5 DGC Overhead

To analyze the extra overhead introduced by DGC, we measure graph partitioning overhead (including chunk generation and chunk assignment) and chunk fusion overhead, as depicted in Figure 4.17. Note that graph partitioning is only invoked once per training job. Due to the substantial gap between different operations, we use a logarithmic function to normalize the overhead. The results indicate that DGC introduces only about 4% overhead to the total training time. Additionally, we find that the Amazon dataset has the lowest chunk generation overhead since it possesses the fewest edges, allowing label propagation to converge swiftly. However, Amazon dataset has a high chunk assignment cost. That is because Amazon dataset has a large number of vertices, resulting in lots of generated chunks and high chunk assignment cost. Although the total chunk fusion overhead exceeds the graph partitioning overhead, it is still significantly less than the total training time. Consequently, the additional overhead introduced by DGC does not significantly detract from the overall performance.

### 4.7.6 Convergence Evaluation

Finally, we study the training convergence of DGC. Figure 4.18 shows the curves of training loss and test accuracy when we train different models on Epinion dataset. We have similar observations on other datasets and thus omit their figures due to space limit, but corresponding accuracy is included in Table 4.3. We can see that all methods, except DGC with stale aggregation, can finally achieve similar accuracy and loss. That



is because all methods adopt full-batch training, without changing training algorithms and related hyper-parameters. The results validate that DGC can guarantee training convergence. In addition, thanks to PGC and the chunk fusion modules, DGC can converge at a faster speed, compared to others. When adaptive stale aggregation module is enabled, DGC can further accelerate the convergence, with minor accuracy degradation.

## 4.8 Summary

This work introduces DGC, a distributed training framework designed to optimize DGNN training efficiency. By incorporating a novel dynamic graph partitioning method (PGC) and run-time optimizations, DGC effectively tackles the challenges of high communication costs and low GPU utilization in distributed DGNN training. Experimental results demonstrate that DGC achieves a  $1.25\times$ - $7.52\times$  speedup compared to state-of-the-art DGNN training frameworks.

# Chapter 5

## Efficient Distributed Graph Learning with Privacy Preserving

### 5.1 Problem Statement

Federated learning has shown great promise in enabling collaborative machine learning among distributed devices while preserving their data privacy [145]. There is a growing amount of research efforts on federated learning [70, 146], but they study CNN models that show superior learning accuracy on image and voice data. However, many applications generate graph data (e.g., social graphs and protein structures) consisting of nodes and edges, and much evidence has shown that Convolutional Neural Network (CNN) cannot efficiently handle graph learning [147, 148]. GCN [21] has been proposed to deal with graph learning by a novel graph convolution operation. Different from CNN's convolution operation that filters a small set of neighboring pixels, a graph convolution operation filters the features of neighboring nodes. Unfortunately, existing work of federated learning mainly focuses on CNN, leaving GCN under explored.

Recently, there are several preliminary research efforts about graph learning on decentralized datasets. Zhou et al. [149] have studied a vertical federated learning scenario on graphs, where clients maintain the same nodes but with different features and edge types. Similarly, Mei et al. [75] assume that graph structural, features and labels belong to different sources. These works are different from the general setting studied in our work. Some recent works explore the intersection of graph and federated learning by discussing the effect of Non-I.I.D data distribution in federated graph learning [150, 151]. However, these works do not consider the inter-graph connections, which is a pervasive phenomenon in the real world [152].

In this work, we study federated learning on GCN based on graph data distributed among multiple computing clients that do not allow direct data sharing due to privacy protection. Each client has a subgraph with edge connections to the subgraphs held by others. Every graph node is associated with some features that contain private information. For example, medical records in hospitals can be organized as graphs, where each graph node represents a record and its features include personal information (e.g., ages, genders, and occupations) as well as health conditions (e.g., diseases) [152]. It has been widely recognized that these feature data is privacy-sensitive and they cannot be exposed. Given some nodes with labels, the goal of graph learning is to predict the labels of other nodes.

Federated learning on GCN is not a simple extension of its counterpart on CNN

because of two unique challenges. First, GCN training involves node feature sharing among clients, leading to the risk of privacy leakage. To exploit graph structure information, the graph convolution operation is designed to aggregate feature data of neighboring nodes. Such an operation would fail if some neighboring nodes are maintained by other clients, who refuse to expose their features. A straightforward solution for privacy protection is to eliminate feature sharing, but it would seriously decrease training accuracy, which has been confirmed by our experimental results. The second challenge is the high training overhead incurred by large graph size [122, 153]. For example, a social network maintained by Facebook contains over 3 billion users, and the corresponding graph data size may be several hundreds of gigabytes [154]. Since a GCN model stacks several layers of the same structure with the original graph, the model size becomes extremely large, even exceeding the physical memory constraint.

In this work, we propose FedGraph, a federated graph learning system that integrates the ideas of federated learning and GCN to open new opportunities for privacy-preserving distributed graph learning. FedGraph is especially good at learning on distributed graphs with complicated connections, and can converge to a high training accuracy by addressing the above challenges. For the first challenge about the dilemma of feature sharing and privacy protection, a common solution is to use cryptography-based techniques, e.g., homomorphic encryption [155, 156], to enable computation over encrypted data. Despite strong security guarantee, these techniques have high computational overhead, making them inappropriate choices for FedGraph that pursues high training speed. There also exist hardware-based solutions, e.g., SGX [157, 158], for privacy protection, but security hardware has limited capacity and it cannot handle large graph data [158]. FedGraph solves the dilemma by designing a cross-client graph convolution operation, without heavy cryptographic operations or dedicated hardware. Instead of directly sharing node features, FedGraph embeds them into low-dimensional representations before sharing, so that original features cannot be recovered.

To reduce GCN training overhead, graph sampling has been widely adopted to randomly select a mini-batch of nodes for training [22, 120, 122, 159]. GraphSAGE [122] is a graph sampling method based on node neighboring relationship. It randomly selects a fixed number of neighbors when applying the graph convolution operation for each node. FastGCN [120] has been proposed to improve sampling efficiency by independently selecting nodes for each graph convolution layer. However, existing work cannot satisfy the requirements of FedGraph design due to three weaknesses. First, these sampling methods depend on some hand-crafted parameters that rely heavily upon the knowledge of domain experts. For example, the performance of GraphSAGE is determined by the parameter specifying the number of sampled neighbors, and manual parameter tuning is time-consuming. Second, existing methods ignore the tradeoff between training speed and training accuracy. Sampling fewer nodes accelerates training but decreases accuracy. Third, clients participating in federated graph learning are heterogeneous in graph size and computational capability. Applying the same sampling policy for all clients is far from the optimal solution.

These weaknesses make the sampling algorithm design challenging in FedGraph. Instead of struggling to improve existing heuristic designs, we resort to DRL techniques and design an intelligent sampling algorithm that can automatically adjust sampling policies by jointly considering computation overhead, training accuracy and client heterogeneity. By carefully examining various DRL algorithms, we choose the Deep Deterministic Policy Gradient (DDPG) and cast it to federated graph learning. The

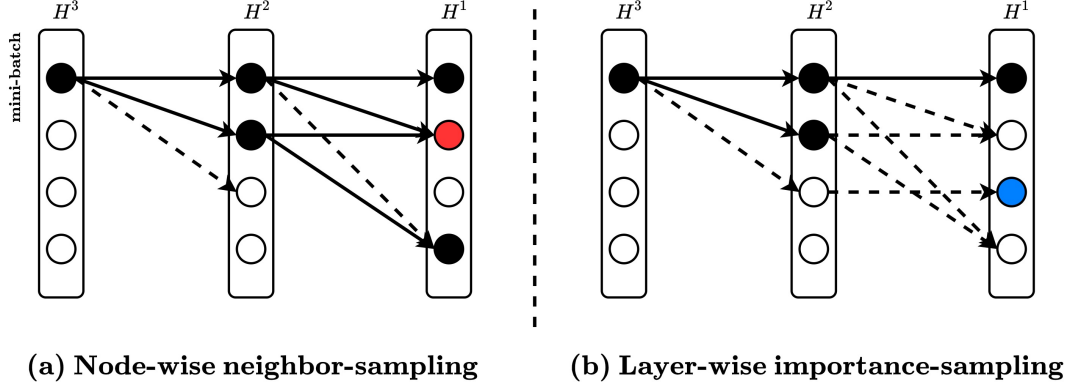


Figure 5.1: An illustration of different sampling approaches. The sampled nodes are marked in color (dark, red, and blue). The dashed arrows denote edge connections in the original graph. The solid arrows denote the edges preserved by sampled nodes.

main contributions of this work are as follows.

1. We propose FedGraph as a novel federated graph learning system. We formally present the procedures of local training by clients and global parameter update by the server. A lightweight cross-client convolution operation is proposed to enable feature sharing among clients while avoiding privacy leakage.
2. A DRL-based sampling algorithm is designed for FedGraph, so that it can automatically find the best sampling policy that makes a good tradeoff between training speed and accuracy.
3. We implement a prototype of FedGraph and evaluate it on a testbed. Four popular graph datasets are used in performance evaluation. The experimental results show that FedGraph enables at least 2 times faster convergence to about 10% higher accuracy than existing work.

## 5.2 Motivation

In many applications, graphs are very large and the corresponding GCN training has high computational overhead. Graph sampling has been proposed to reduce the sizes of graphs used for GCN training, and its existing work can be classified into two categories. One is node-wise neighbor-sampling that iteratively samples a fixed number of neighbors for each node. As shown in Fig. 5.1(a), given some nodes in the  $(l + 1)$ -th layer, we randomly select a subset of their neighbors as the  $l$ -th layer. Such a sampling guarantees that aggregation of node embeddings always happens among neighboring nodes. A representative work of node-wise neighbor-sampling is GraphSAGE [122]. However, the number of sampled nodes may exponentially increase as more layers are constructed. In addition, Huang et al. [160] have pointed out that it incurs redundancy of embedding calculation at some nodes, e.g., the red nodes in Fig. 5.1(a), which are the shared neighbors of other nodes. Several recent approaches, e.g., VR-GCN [159] and Cluster-GCN [10], have been proposed to improve the performance of node-wise neighbor-sampling, but they cannot fundamentally address this weakness.

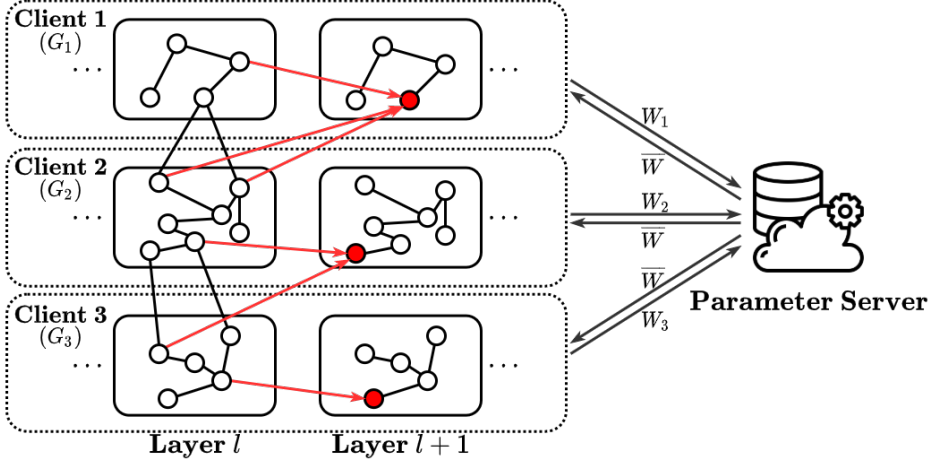


Figure 5.2: The FedGraph architecture. Each client  $i$  maintains a local graph  $G_i$ . During the training, nodes in the mini-batch (nodes in red) aggregate neighbors' embeddings to generate the next layer's embeddings, denoted by red arrows. When training completes, each client  $i$  uploads its local model weights  $W_i$  to the parameter server. Finally, the parameter server aggregates all local model weights to the updated global model  $\bar{W}$  and sends it back to all clients.

The other kind of approaches is called layer-wise importance-sampling. Its basic idea is to independently sample a fixed number of nodes for each GCN layer based on a sampling probability, which is calculated based on node degrees. FastGCN [120] is a typical approach of layer-wise importance-sampling. However, since nodes of different layers are sampled independently, some sampled nodes may have no connections with the ones in the previous layer, like the blue-marked node shown in Fig. 5.1(b). The embeddings of some unlinked nodes may be lost during graph convolution operations, which would deteriorate the training performance.

The strengths and weaknesses of both sampling approaches motivate us to design a new sampling policy that can well control the computation overhead while keeping neighboring relations during sampling.

### 5.3 FedGraph Design

We consider a typical setting of federated graph learning, which consists of a set  $C$  of computing clients that conduct local training tasks, and a server responsible for global parameter update, as shown in Fig. 5.2. Computing clients and the server may locate at different locations and they are connected by wide-area networks. Each client  $i \in C$  maintains a graph  $G_i(V_i, E_i)$ , where each node  $v \in V_i$  is associated with a feature vector  $x(v)$  that cannot be exposed to other clients. A subset  $V_i^{label} \subseteq V_i$  of nodes have labels denoted by  $\{y(v) | v \in V_i^{label}\}$ , which can be used as training data. The edge set  $E_i$  contains the internal edges among nodes in  $V_i$ , as well as the external ones connecting to nodes held by other clients. Each client is aware of the existence of neighboring nodes maintained by others but cannot directly access their feature vectors.

We assume that computing clients and the parameter server are *honest-but-curious*, i.e., they honestly follow the federated learning procedures but want to learn feature

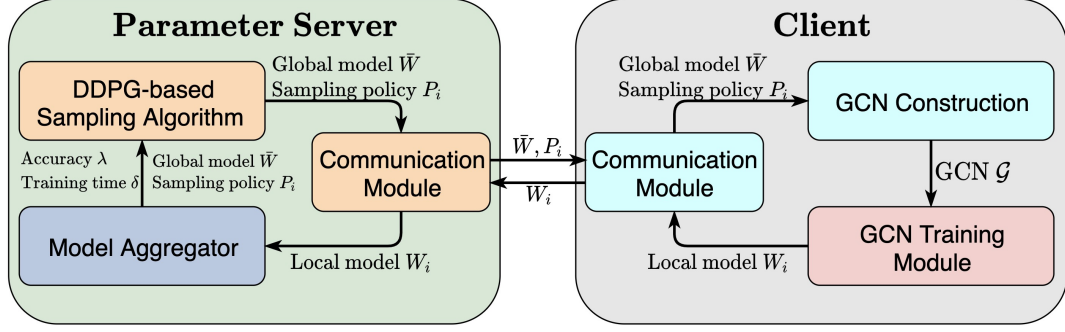


Figure 5.3: System design.

information held by others. This is a typical threat model that has been widely used by current federated learning research [155, 161, 162]. Some other more serious threat models are discussed as follows. Some malicious clients can tamper with the training by modifying model parameters sent to the parameter server. To deal with this threat, we can use Trusted Execution Environment (TEE) for local training. TEE is commonly available on modern CPUs. It enables an isolated execution environment guaranteed by hardware, and adversaries cannot access data and codes in TEE. Besides, malicious parameter servers can modify global model parameters to compromise federated learning. We can use secure Multi-party Computation (MPC) or Homomorphic Encryption (HE) to protect the model aggregation. Besides, TEE can be also used to protect global model aggregation at the parameter server.

Our system design is shown in Fig. 5.3. We customize the parameter server and clients by adding new modules to implement intelligent sampling. The parameter server contains three main modules. The DDPG-based sampling algorithm generates sampling policies for all clients. A model aggregator collects local feature weights trained by clients and aggregates them to generate new global feature weights for next-round training. In addition, a communication module is designed for message exchanges between the parameter server and clients. This communication module is realized by gRPC APIs, which are based on TCP communication protocol. Each client has a module of GCN construction, responsible for creating a GCN model according to sampling policy. A GCN training module is designed to run the training algorithm.

In FedGraph, in order to predict  $y(v)$  of unlabeled nodes, clients collaboratively train global feature weights  $\bar{W}$ . There are multiple training rounds. In each round, clients download the latest feature weights from the server and construct local GCNs to train these weights. Due to the existence of external edge connections, local GCN training involves embedding sharing among clients. After that, they send updated feature weights to the server, which then creates new global feature weights that will be used for the next-round training. Although FedGraph shares a similar process with traditional federated learning, it has unique procedures of local training and global parameter update, which are presented as follows.

### 5.3.1 Local GCN Training by Clients

The local GCN training procedure of each client  $i \in C$  is described in the Algorithm 3. At the beginning of each round, client  $i$  downloads the latest feature weights  $\bar{W}$  as well as a graph sampling policy  $P_i$  from the server. The local feature weights are

initialized as  $W_i = \overline{W}$ . Then, this client launches multiple training iterations to update feature weights based on local graph data. Specifically, each training iteration consists of the following two main steps.

---

**Algorithm 3** Local training procedure of client  $i \in C$

---

- 1: **for** each training round  $t$  **do**
- 2:   Download the latest feature weights  $\overline{W}$  and a sampling policy  $P_i$  from the parameter server;
- 3:   Initialize the local feature weights as  $W_i = \overline{W}$ ;
- 4:   **for** each iteration **do**
- 5:     Construct a GCN  $\mathcal{G}_i = \{\mathcal{V}_i^{(1)}, \mathcal{V}_i^{(2)}, \dots, \mathcal{V}_i^{(L)}\} = \mathbf{ModelConstruct}(G_i, P_i)$
- 6:     **for** each layer  $l = 1, 2, \dots, L - 1$  **do**
- 7:       **for** each node  $v \in \mathcal{V}_i^{(l)}$  **do**
- 8:         **if**  $l = 1$  **then**
- 9:

$$z_i^{(l+1)}(v) = \sum_{u \in V_i} \tilde{Q}_i^{(l)}(v, u) h_i^{(l)}(u) W_i^{(l)}; \quad (5.1)$$

- 10:       **else if**  $l > 1$  **then**
- 11:

$$\begin{aligned} z_i^{(l+1)}(v) = & \sum_{u \in V_i} \tilde{Q}_i^{(l)}(v, u) h_i^{(l)}(u) W_i^{(l)} + \\ & \sum_{j \neq i} \sum_{u \in V_j} \tilde{Q}_i^{(l)}(v, u) h_j^{(l)}(u) W_j^{(l)}; \end{aligned} \quad (5.2)$$

- 12:       **end if**
- 13:     Generate the embeddings of the  $(l + 1) - th$  layer:

$$h_i^{(l+1)}(v) = \sigma(z_i^{(l+1)}(v)); \quad (5.3)$$

- 14:     **end for**
- 15:   **end for**
- 16:   Calculate the loss according to the function:

$$\mathcal{L} = \frac{1}{|\mathcal{V}_i^{(L)}|} \sum_{v \in \mathcal{V}_i^{(L)}} \text{loss}(y(v), z_i^L(v)) \quad (5.4)$$

- 17:   Update the local feature weight:

$$W_i \leftarrow W_i - \epsilon \nabla \mathcal{L} \quad (5.5)$$

- 18:   **end for**
  - 19:   Submit updated feature weights  $W_i$  to the server;
  - 20: **end for**=0
-

### GCN construction

---

**Algorithm 4** The pseudocodes of **ModelConstruct()**

---

- 1: Randomly select  $\kappa_i$  labeled nodes as a mini-batch and include them into the  $L$ -th layer, i.e.,  $\mathcal{V}_i^{(L)}$ ;
- 2: **for** each layer  $l = L - 1, \dots, 2, 1$  **do**
- 3:   **for** each node  $v \in \mathcal{V}_i^{(l+1)}$  **do**
- 4:     Sample a subset  $N_i^{(l)}(v)$  of neighbors according to a selection probability  $p_i^{(l)}$ ;
- 5:     Update the adjacent matrix  $\tilde{Q}_i^{(l)}$  as follows.

$$\tilde{Q}_i^{(l)}(v, u) = \begin{cases} \frac{|V_i(v)|}{|N_i^{(l)}(v)|} Q_i(v, u), & \text{if } u \in N_i^{(l)}(v); \\ 0, & \text{otherwise;} \end{cases} \quad (5.6)$$

- 6:   **end for**
  - 7:    $\mathcal{V}_i^{(l)} = \cup_{v \in \mathcal{V}_i^{(l+1)}} N_i^{(l)}(v)$ ;
  - 8: **end for=0**
- 

We construct a GCN  $\mathcal{G}_i$  of  $L$  layers, using the function **ModelConstruct()** that samples a subset of nodes according to the policy  $P_i$ . The basic idea is to start by randomly selecting a set of nodes with labels, which is also referred to as a mini-batch. For each node in the mini-batch, we then iteratively aggregate the embeddings of a sampled subset of neighbors at most  $L - 1$  hops away.

The pseudo codes of **ModelConstruct()** are shown in Algorithm 4. Specifically, a sampling policy can be expressed by  $P_i = \{\kappa_i, p_i^{(1)}, p_i^{(2)}, \dots, p_i^{(L-1)}\}$ , where  $\kappa_i$  denotes the mini-batch size, and  $\{p_i^{(1)}, p_i^{(2)}, \dots, p_i^{(L-1)}\}$  are neighbor sampling probabilities of  $L - 1$  layers, respectively. As shown in line 1, we sample  $\kappa_i$  labeled nodes as the mini-batch and they compose the final  $L$ -th layer. Then, we iteratively construct other GCN layers in a backward direction. For each node  $v$  in the  $(l + 1)$ -th layer, we randomly select a subset  $N_i^{(l)}(v)$  of its neighbors into the  $l$ -th layer with a probability  $p_i^{(l)}$ . In addition, we create a matrix  $\tilde{Q}_i^{(l)}$  to replace  $Q$  in (5.6), where  $V_i(v)$  denotes the set of neighbors of node  $v$  in the original graph  $G_i$ . The matrix  $\tilde{Q}_i^{(l)}$  describes the updated adjacent relation after sampling, and it will be used for feature aggregation later. All sampled nodes in the  $l$ -th layer are maintained in set  $\mathcal{V}_i^{(l)}$ , as shown in the final line.

The GCN construction combines the strength of node-wise sampling and layer-wise sampling. These sampling probabilities are independent, which offers opportunities for fine-grained sampling over layers, like layer-wise sampling. By carefully setting these probabilities, we can avoid the high computational cost incurred by the recursive explosive expansion of the neighborhood. Meanwhile, since the sampling process is based on neighborhood relation, which is similar to node-wise sampling, we can avoid sampling nodes without connections.

### GCN training

After constructing a GCN model, we continue to train this GCN based on gradient descent. The cross-client graph convolution operation is described in lines 7-13 of Algorithm 3. Specifically, clients aggregate embeddings of only internal neighbors



when they process the first GCN layer, as shown in Eq. (5.1). From the second layer, we enable clients to aggregate both internal neighbors and external ones, which is shown in Eq. (5.2). Such a design can prevent the leakage of local origin features while enabling information sharing. We will give the security analysis in Section 5.3.3.

After aggregation, a nonlinear transformation is applied to generate the node embedding  $h_i^{(l+1)}(v)$  of the next layer, as shown in Eq. (5.3). With the objective of minimizing a loss function defined in Eq. (5.4), we compute the gradients and update feature weights in Eq. (5.5), where  $\epsilon$  is the learning rate. Finally, client  $i$  submits the updated feature weights (or their differences from downloaded ones) to the parameter server.

### 5.3.2 Global Parameter Update by the Server

---

**Algorithm 5** Global weight update of parameter server

---

- 1: Initialize random feature weights  $\bar{W}$  and sampling policies  $\{P_1, P_2, \dots, P_{|C|}\}$ , and send them to clients, respectively;
- 2: **for** each training round  $t$  **do**
- 3:   Collect feature weights  $\{\bar{W}, W_1, W_2, \dots, W_{|C|}\}$ , from all clients;
- 4:   Create global feature weights:

$$\bar{W} = \sum_{i \in C} \frac{\kappa_i}{\sum_{i \in C} \kappa_i} W_i; \quad (5.7)$$

- 5:   Update the sampling policy  $\{P_1, P_2, \dots, P_{|C|}\} = \text{GenSampling}(\bar{W}, W_1, W_2, \dots, W_{|C|})$ ;
  - 6:   Send global feature weights  $\bar{W}$  and sampling policy  $P_i$  to every client  $i \in C$ ;
  - 7: **end for**
- 

The procedure of global weight update by the parameter server is shown in Algorithm 5. The server starts by initializing random feature weights  $\bar{W}$  and sampling policies  $\{P_1, P_2, \dots, P_{|C|}\}$ , and then sends them to clients, respectively. In each of the following training rounds, it collects updated local feature weights from all clients, followed by two main tasks. First, it creates global feature weights by aggregating local weights as shown in Eq. (5.7), where  $\kappa_i$  denotes the mini-batch size, i.e., the number of labeled nodes, at client  $i$  in the current training round. The second task is to update sampling policies for clients using function **GenSampling()**, whose details will be given in the next section. The design of **GenSampling()** is one of the most important contributions of this work, and it relies on the deep reinforcement learning technique to balance computational overhead and model accuracy. Finally, the server sends new global feature weights and sampling policies to clients to start the next round of training.

### 5.3.3 Security Analysis

To show how our proposed Algorithm 3 protects feature data, we consider two clients  $i$  and  $j$ , who need to share node embeddings during training, without loss of generality. Suppose client  $i$  aggregates embeddings from client  $j$  and wants to infer the

original node features  $h_j^{(1)}$ . Note that  $h_j^{(1)}$  is a matrix containing features of all nodes held by client  $j$ , i.e.,  $h_j^{(1)}(v) = x_j(v)$ ,  $v \in V_j$ .

We let  $V_j^i$  denote the client  $i$ 's neighboring nodes at client  $j$ . According to Algorithm 3, client  $i$  can get information of  $\{h_j^{(2)}(V_j^i)W_j^{(2)}, h_j^{(3)}(V_j^i)W_j^{(3)}, \dots, h_j^{(L)}(V_j^i)W_j^{(L)}\}$ . Then, client  $i$  can guess node embeddings  $\{h_j^{(2)}, \dots, h_j^{(L)}\}$  by approximating remote  $W_j^{(l)}$  using local  $W_i^{(l)}$ , which is possible when they just synchronize global feature weights from the server.

However, it would be difficult for client  $i$  to further infer  $h_j^{(1)}(V_j^i)$  because  $h_j^{(2)} = \sigma(\tilde{Q}_j^{(1)}h_j^{(1)}W_j^{(1)})$  and client  $i$  has no information about  $\tilde{Q}_j^{(1)}$ , i.e., the adjacent matrix in client  $j$  after sampling. Furthermore, the guess of  $\{h_j^{(2)}, \dots, h_j^{(L)}\}$  can hardly achieve high accuracy due to the dimension reduction of embeddings in higher layers. Given that original features of neighboring nodes can be protected, it would be impossible to get the features of internal nodes at client  $j$ . Therefore, we can conclude that FedGraph can protect the node features while enabling information sharing during federated graph learning.

## 5.4 Intelligent Graph Sampling based on DRL

Sampling policies  $\{P_1, P_2, \dots, P_{|C|}\}$  determine how many nodes are involved in GCN training, and they affect both computational overhead and training accuracy. By sampling fewer nodes, we can accelerate the training process with reduced computational overhead, while lowering training accuracy. On the other hand, with more sampled nodes, we can better approximate the original GCN to achieve higher training accuracy, but incurs a high computational cost. Therefore, it is significant to design sampling policies to make a tradeoff, however, which has been ignored by existing work. Meanwhile, sampling policy design is difficult due to a large optimization space, and manual tuning hardly works in practice. We desire automatic algorithms, with minimum human involvement, to generate good sampling policies.

By carefully examining sampling policies, we find that their influence on the learning performance, in terms of training speed and accuracy, cannot be described using precise closed-form expressions. Instead of struggling with heuristic algorithm design, we resort to DRL that can automatically approximate a good solution. The idea of DRL can be implemented in various ways, generating a thriving family of algorithms for different application scenarios with different performance. By carefully comparing candidate DRL algorithms, we choose to use DDPG algorithm [163], which can efficiently handle the high-dimensional and continuous action space of our problem. DDPG combines Deep Q-Networks and actor-critic approach and thus enjoys their benefits.

### 5.4.1 DDPG-based problem formulation

To apply DDPG, we first formulate our problem as a Markov decision process as follows.

**State Space:** We define the system state of the training round  $t$  as the observed feature weights at the beginning of this round, which can be represented by  $s[t] = \{\bar{W}[t], W_1[t], W_2[t], \dots, W_{|C|}[t]\}$ . Note that  $\bar{W}[t]$  is the global feature weights and  $W_i[t]$  denotes the local feature weights of client  $i \in C$ . The whole action space is denoted

by  $\mathcal{S}$ . Since the state space is huge, we leverage the principal component analysis (PCA) [164] to project the high-dimensional space onto a lower-dimensional space while keeping the distribution information as complete as possible.

**Action Space:** At the beginning of round  $t$ , the parameter server needs to decide graph sampling policies for all clients. The action  $a[t]$  of each round  $t$  is therefore defined as the corresponding sampling policies, i.e.,  $a[t] = \{P_1[t], P_2[t], \dots, P_{|C|}[t]\}$ . The action space is denoted by  $\mathcal{A}$ .

**Reward:** Since both learning speed and accuracy are considered as performance metrics, the reward should be defined to reflect them. We use the completion time of each training round  $t$ , which is denoted by  $\delta[t]$ , to evaluate the training speed. The server can easily obtain  $\delta[t]$  by measuring the time consumption of collecting local training results from all clients. The training accuracy  $\lambda[t]$  is calculated based on a testing set at the parameter server. We consider a typical federated setting, where the parameter server is usually the task publisher that holds a testing set. Each client has its own training set and validation set, which cannot be exposed due to privacy concerns. With the information of  $\delta[t]$  and  $\lambda[t]$ , we define the reward of each round  $t$  as follows,

$$r[t] = \Omega^{(\lambda[t] - \Lambda)} - \alpha(\delta[t] - \beta), \quad (5.8)$$

where  $\Lambda$  is the target accuracy. The constants  $\Omega$ ,  $\alpha$  and  $\beta$  can be adjusted to express different preferences on learning speed and accuracy. The reward contains two parts. The first part evaluates accuracy improvement. We notice that  $\lambda[t]$  shows nonlinear improvements as the learning proceeds. It can be quickly improved in the first few training rounds, but the improvement becomes smaller later. In order to make the reward unbiased, we use an exponential function here. The second part evaluates the completion time of each training round in the negative form, to encourage fast training. In practice, the completion time of a client is affected by many factors, i.e., computational hardware or network latency. We alleviate the impact of these factors by adding a constant  $\beta$  in (5.8), so that we can better evaluate the influence of different sampling policies. In our experiments, we control the time penalty, i.e.,  $\alpha(\delta[t] - \beta)$ , close to 1, as referred to [72], which can be easily achieved by profiling.

**Learning policy and objective:** We define the DRL learning policy in our problem as  $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ , which is parameterized by  $\theta$ . More precisely, given a state  $s[t]$ , the algorithm outputs a deterministic action  $a_t$ . The objective of our DRL-based sampling algorithm is to maximize the expected cumulative discounted reward from the starting state, which is defined as:

$$J(\theta) = \mathbb{E}[R[t] | S[t] = s[t]],$$

where  $R[t] = \sum_{k=0}^{\infty} \gamma^k r[t+k]$  is the cumulative discounted reward function.

The action-value function  $q_\pi(s[t], a[t])$  is defined to describe the expected cumulative discounted reward after executing action  $a[t]$  in state  $s[t]$  based on policy  $\pi_\theta$ , i.e.,  $q_\pi(s[t], a[t]) = \mathbb{E}[R[t] | S[t] = s[t], A[t] = \pi_\theta(s[t])]$ . Typically, we use neural networks to approximate the policy function  $\pi_\theta$  and action-value function  $q_\pi$ .

### 5.4.2 Sampling based on DDPG

The DDPG-based sampling algorithm design is illustrated in Fig. 5.4. We design an actor network  $\mu(s|\theta_\mu)$  to predict deterministic actions, and a critic network  $q(s, a|\theta_q)$  to

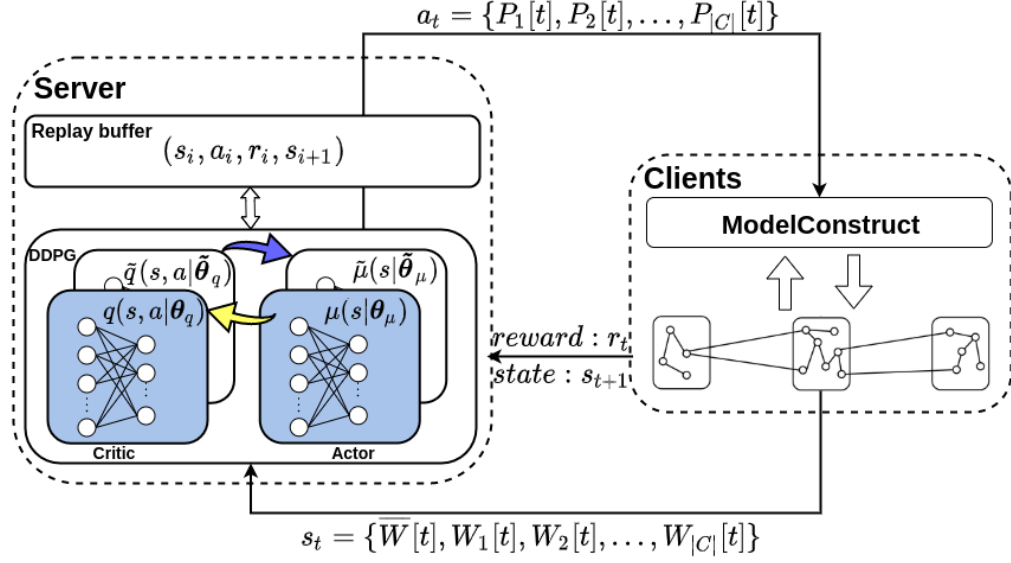


Figure 5.4: Illustration of DDPG-based sampling.

estimate the action-value function  $q_\pi(s, a)$ . Meanwhile, we maintain copies of the actor network and critic network, denoted by  $\tilde{\mu}(s | \tilde{\theta}_\mu)$  and  $\tilde{q}(s, a | \tilde{\theta}_q)$ , which are also referred to as target networks. They can be used to update the original actor and critic networks.

Similar to Deep Q-Networks, we maintain a replay buffer of finite size to store historical transitions defined as  $(s[t], a[t], r[t], s[t+1])$ . We update the actor and critic networks by sampling a mini-batch of transitions from the replay buffer. When the buffer is full, the oldest samples are discarded. We then formally introduce the DRL-based sampling algorithm, i.e., implementation details of function **GenSampling()**, and explain how it learns the optimal sampling scheme.

The pseudo codes of the DDPG-based algorithm are shown in Algorithm (6). We initialize four networks as well as the system state in lines 1–5. At the beginning of training round  $t$ , the server observes the current state information  $s[t]$  in the form of feature weights of all clients, and the reward  $r[t-1]$  defined in (5.8), as shown in line 9. Then, we reduce the dimension of  $s[t]$  to get  $s'[t]$  using the PCA method [164], and then store the transition  $(s'[t-1], a[t-1], r[t-1], s'[t])$  into the replay buffer. After that, we randomly select a mini-batch of  $K$  transitions to update the critic network by minimizing the loss function:

$$\mathcal{L} = \frac{1}{K} \sum_{k=1}^K (q^{target} - q(s'[t_k - 1], a[t_k - 1] | \theta_q))^2, \quad (5.9)$$

where  $q^{target} = r[t_k - 1] + \gamma \tilde{q}(s'[t_k], \tilde{\mu}(s'[t_k] | \tilde{\theta}_\mu) | \tilde{\theta}_q)$  is the target action value. The parameters of the critic network are updated by:

$$\theta_q[t] = \theta_q[t-1] - \eta_q \nabla \mathcal{L}, \quad (5.10)$$

where  $\eta_q$  is the learning rate. Then we update the actor network as follows:

$$\begin{aligned} \boldsymbol{\theta}_\mu[t] = & \boldsymbol{\theta}_\mu[t-1] - \\ & \eta_\mu \left[ \frac{1}{K} \sum_i \nabla_a q(s[i], a) \nabla_{\boldsymbol{\theta}_\mu} \mu(s[i])|_{a=\mu(s[i])} \right], \end{aligned} \quad (5.11)$$

where  $\eta_\mu$  is the learning rate of the actor network. The parameters of two target networks are updated in line 14, where  $\phi \ll 1$ . Finally, we obtain the action  $a[t]$  representing sampling policies based on updated networks.

---

**Algorithm 6** Sampling Algorithm Based on DRL
 

---

- 1: Randomly initialize the actor  $\mu(s|\boldsymbol{\theta}_\mu)$  and critic  $q(s, a|\boldsymbol{\theta}_q)$  with parameters  $\boldsymbol{\theta}_\mu$  and  $\boldsymbol{\theta}_q$ ;
- 2: Initialize the target networks  $\tilde{\mu}(s|\tilde{\boldsymbol{\theta}}_\mu)$  and  $\tilde{q}(s, a|\tilde{\boldsymbol{\theta}}_q)$  with parameters  $\tilde{\boldsymbol{\theta}}_\mu \leftarrow \boldsymbol{\theta}_\mu$  and  $\tilde{\boldsymbol{\theta}}_q \leftarrow \boldsymbol{\theta}_q$ ;
- 3: Initialize the initial state  $s[0] = \{\bar{W}[0], W_1[0], \dots, W_{|C|}[0]\}$ ;
- 4: Reduce the dimension of initial state:  $s'[0] = \mathbf{PCA}(s[0])$ ;
- 5: Initialize the exploration noise  $\Delta$  and replay buffer;
- 6: Generate sampling policies represented by  $a[0] = \mu(s'[0]|\boldsymbol{\theta}_\mu) + \Delta_0$  and send them to clients;
- 7: **for** episode = 1, 2, ...,  $Z$  **do**
- 8:   **for**  $t = 1, 2, \dots, T$  **do**
- 9:     Observer the state  $s[t]$  and reward  $r[t-1]$ ;
- 10:     $s'[t] = \mathbf{PCA}(s[t])$ ;
- 11:    Store the transition  $(s'[t-1], a[t-1], r[t-1], s'[t])$  into the replay buffer;
- 12:    Randomly select a mini-batch of  $K$  transitions from the replay buffer;
- 13:    Update the critic and actor networks by (5.10) and (5.11);
- 14:    Update the target networks by soft update method:

$$\tilde{\boldsymbol{\theta}}_\mu = \phi \boldsymbol{\theta}_\mu + (1 - \phi) \tilde{\boldsymbol{\theta}}_\mu, \quad (5.12)$$

$$\tilde{\boldsymbol{\theta}}_q = \phi \boldsymbol{\theta}_q + (1 - \phi) \tilde{\boldsymbol{\theta}}_q; \quad (5.13)$$

- 15:   Generate sampling policies  $a[t] = \mu(s'[t]|\boldsymbol{\theta}_\mu) + \Delta_t$ ;
  - 16:   **end for**
  - 17: **end for**=0
- 

## 5.5 Evaluation

### 5.5.1 Experimental settings

We implement FedGraph using PyTorch and Deep Graph Library (DGL) [165], a Python package dedicated to deep learning on graphs. We deploy FedGraph on 20 computing clients with Intel i7-10700 CPU, 32GB memory, and Geforce RTX 2080 GPU. We consider 4 popular graph datasets: Cora, Citeseer, PubMed, and Reddit, which have been widely used for GCN studies [10, 22, 120, 122, 159, 160]. Some statistic information of these datasets is summarized in Table 5.1. Since some graphs, e.g., Cora and

Table 5.1: Graph Data Statistics

Dataset	Nodes	Edges	Features	Classes
Cora	2,708	10,556	1,433	7
Citeseer	3,327	9,228	3,703	6
PubMed	19,717	88,651	500	3
Reddit	232,965	114,848,857	602	41

Citeseer, are with limited sizes, we synthesize large graphs based on these datasets using the following method. Given a dataset in Table 5.1, each client  $i$  randomly selects a proportion  $\xi_i$  of nodes as its local graph data, and  $\{\xi_1, \xi_2, \dots, \xi_{|C|}\}$  belongs to a normal distribution with a mean of 0.8. It is possible that generated local graphs overlap on some nodes, especially for small graph datasets, like Cora and Citeseer. For large graphs, we carefully control the local graph generation to avoid overlapping. Even some nodes overlap in the synthesized datasets, we treat them as different nodes and there is no influence to training performance. A similar graph synthesis method has been adopted by [25]. For the local dataset, we randomly choose a set of nodes to generate a training set, a validation set, and a test set. The edge connections across clients are maintained according to the original graph. For local graph learning, each client constructs a 3-layer GCN, including an input layer and two convolutional layers. We set 16 hidden units, 50% dropout rate, 0.01 learning rate for Cora, Citeseer, and PubMed. For Reddit, there are 128 hidden units, the dropout rate is 20%, and the learning rate is 0.0001. We set the batch size as 256 for Cora, Citeseer, and Reddit, 1024 for PubMed [120]. We use ADAM optimizer for local GCN training. For the reward function (5.8), we set the base of exponential function, i.e.,  $\Omega$ , as 128 in our experiments. Since FedGraph relies on the exponential property of reward function, the base has little influence on FedGraph. Moreover, the difference of training accuracy  $\lambda[t]$  and target accuracy  $\Lambda$  affects the reward in each round  $t$ . For each dataset, we choose the best accuracy reported by existing work. Even we have no knowledge of the best accuracy, we can make an estimation according to experiences. Since FedGraph only relies on the exponential property of reward function, such estimation has little influence to FedGraph. Both constants  $\alpha$  and  $\beta$  aim to balance accuracy improvement and time cost. In our experiments, we control the time penalty  $\alpha(\delta[t] - \beta)$  close to 1, similar to the settings in [72]. For comparison, we extend the following three graph sampling schemes for federated graph learning.

1. **Full-batch:** We do not conduct graph sampling and use the original graph to construct GCN.
2. **GraphSAGE:** A typical node-wise neighbor-sampling method that iteratively samples a fixed number of neighbors. The neighbor-sampling sizes of two convolutional layers are set as 25 and 10, respectively, which are the same with the settings in [10, 120, 122].
3. **FastGCN:** A typical layer-wise importance-sampling method that independently samples a fixed number of nodes, which is also called layer size, for each layer. The layer size of Cora and Citeseer is set to 256, and that of Reddit and PubMed is 8192, which are the settings advocated by [22].

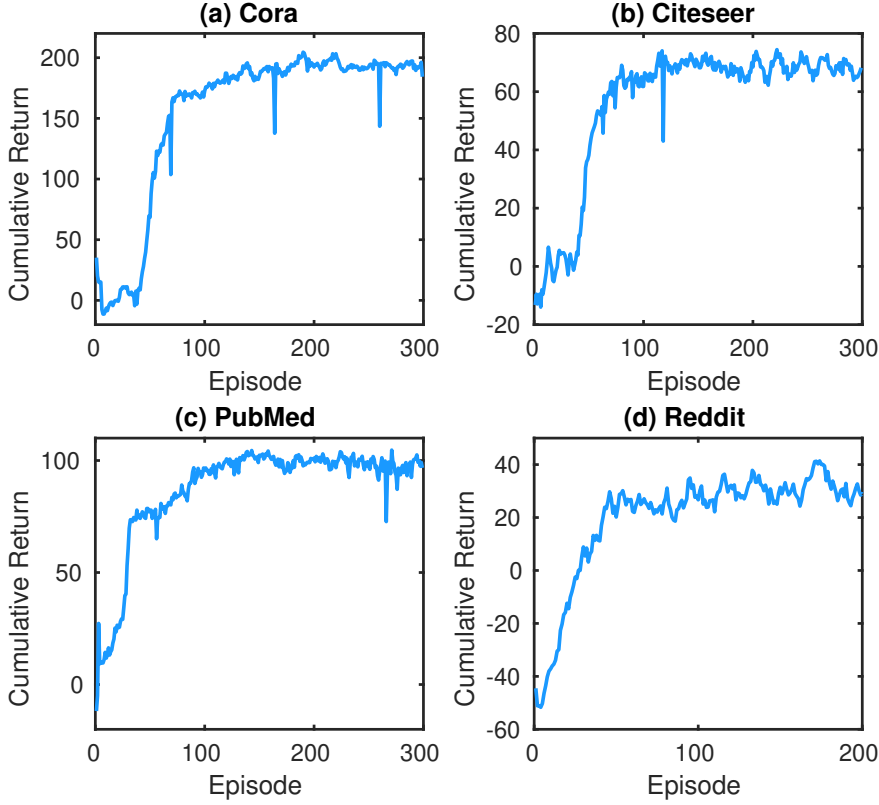


Figure 5.5: Cumulative discounted returns of FedGraph.

In the DRL-based sampling algorithm of FedGraph, both actor-networks and critic-networks have 2 hidden layers of 512 and 256 units. We compress feature weights into 20 dimensions by using the tool `sklearn.decomposition.PCA` [166].

### 5.5.2 Experimental results

**Convergence of DRL-based sampling.** We let FedGraph train 300 episodes and show cumulative returns under four datasets in Fig. 5.5. We set the target accuracy as 90.16% for Cora, 78.7% for PubMed, 87.9% for Citeseer, and 96.27% for Reddit. We observe that cumulative discounted returns of four datasets can converge to stable values in less than 100 episodes, Especially, the biggest dataset, Reddit, almost converges after 50 episodes, as shown in Fig. 5.5(d). These facts demonstrate good convergence of our proposed DRL-based sampling scheme.

**Results of training accuracy.** The accuracy convergence of different sampling schemes is shown in Fig. 5.6, where we can see that FedGraph can converge at a faster speed and achieve higher accuracy. For a fair comparison, we use physical time, instead of the number of training rounds, as the metric to evaluate training speeds of different schemes. That is because clients have graphs of different sizes, and they consume different time costs in each training round. Specifically, FedGraph achieves 75% accuracy at about 5 seconds on Cora, but the other three algorithms take more than 10 seconds to achieve similar accuracy. In PubMed, FedGraph takes about 15 seconds to achieve 73% accuracy, but GraphSAGE and full-batch scheme need more than 2 times as long

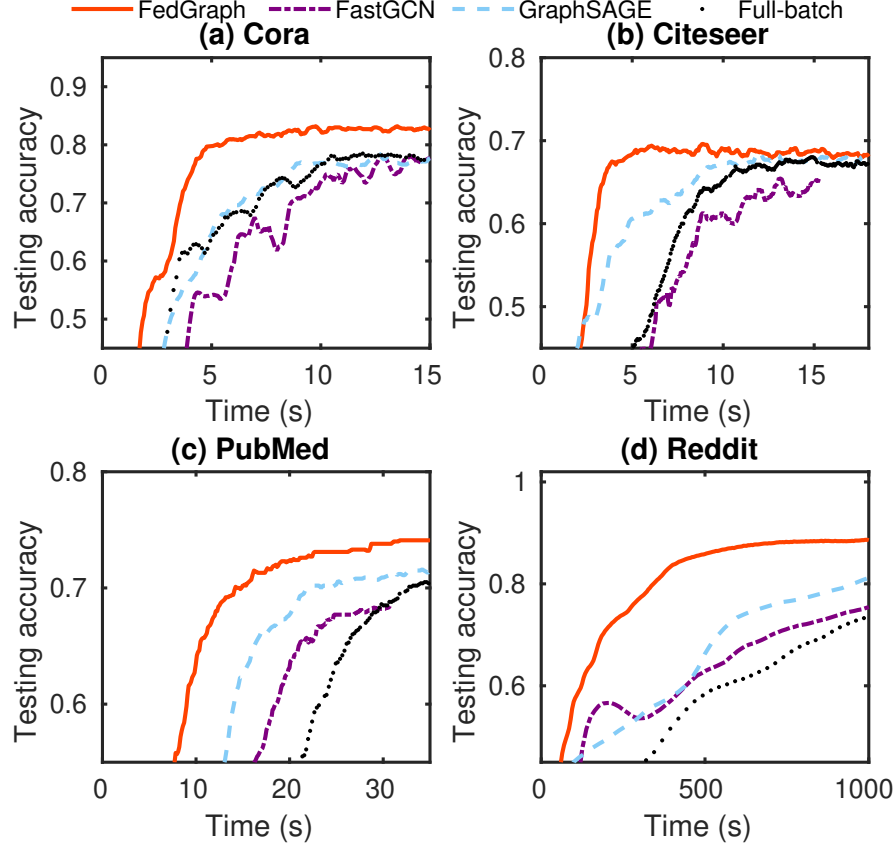


Figure 5.6: Accuracy convergence of different sampling schemes with 20 clients. Note that, FastGCN completes the training with less time as it samples fewer nodes for training. However, it has poor performances in all datasets.

to converge. In the largest datasets Reddit, FedGraph’s advantages are more obvious, as shown in Fig. 5.6(d). We summarize the reasons as follows. GraphSAGE has a serious problem of computation redundancy, which consumes more time for training. FastGCN can not get sufficient embedding information from other clients because some sampled nodes have no edge connections. Full-batch scheme needs to calculate the embeddings of all nodes, which incurs high computational cost especially on larger graphs PubMed and Reddit. FedGraph has well addressed the weaknesses of the above methods and thus achieves higher performance. Note that the total number of training rounds is fixed to 300 and FastGCN completes training earlier because it samples fewer nodes for training. Moreover, to evaluate the scalability of FedGraph, we enlarge the experimental scale to 50 clients and show corresponding results in Fig. 5.7. We can find that FedGraph still outperforms other sampling schemes.

**Influence of graph heterogeneity.** We study the influence of graph heterogeneity by changing the variance of  $\xi_i$ . We consider three heterogeneity levels, and the corresponding variances are 0.1 (low), 0.5 (middle) and 1 (high), respectively. For a better understanding, we calculate the ratio between the smallest graph size and the largest size, and the results are about 0.2, 0.4 and 0.6, respectively. We measure the training time to converge to a target accuracy that can be achieved by most of sampling schemes. In PubMed, we set target accuracy to 72%, but FastGCN can converge to 68.6% only.



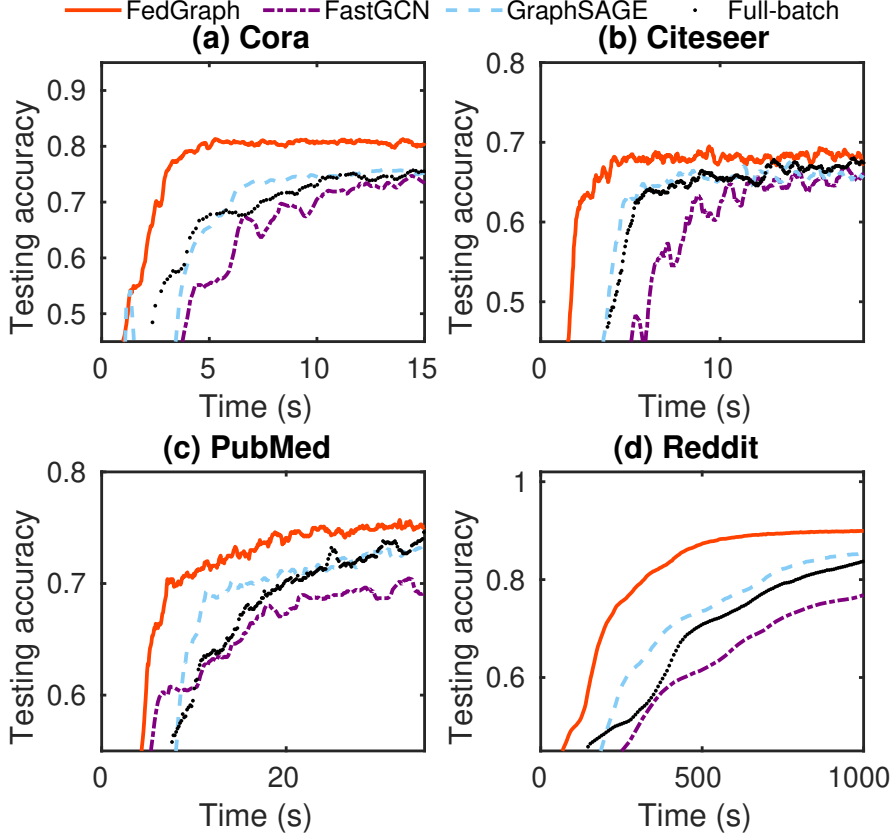


Figure 5.7: Accuracy convergence of different sampling schemes with 50 clients.

As shown in Fig. 5.8, the training time of all sampling schemes increases as graphs become more heterogeneous under all datasets. However, FedGraph has better control on the time growth because its DRL-based sampling jointly considers the training speed and accuracy.

**Effect of cross-client embedding sharing.** FedGraph uses the cross-client graph convolution operation to enable embedding sharing between clients while hiding local features during local GCN training. For comparison, we consider two alternative methods, one (referred to as FedGraph\_allShare) is to share embeddings from the first layer to maximize the information sharing, and the other (referred to as FedGraph\_nonShare) is to discard cross-client sharing to simplify the design. We show the accuracy convergence of these three designs in Fig. 5.9. The total number of training rounds is set to 300. We can find that the curve of FedGraph is close to that of FedGraph\_allShare, which demonstrates that FedGraph has little information loss even though it eliminates the embedding sharing in the first layer. It is because that the high-layer embedding contains information about the original features. Hence, FedGraph can efficiently learn from cross-clients embedding sharing without the original feature exchanging. Simultaneously, FedGraph significantly outperforms FedGraph\_nonShare under all datasets. In Cora and Citeseer, cross-client convolution operations can increase training accuracy by about 10%. In PubMed, two designs have similar final accuracy, but FedGraph enables quick convergence. Reddit is more sensitive to cross-client embedding sharing than other datasets, and FedGraph\_nonShare converges to an accuracy of about 70%, while FedGraph can converge to about 90%. That is because Reddit has rich edge con-

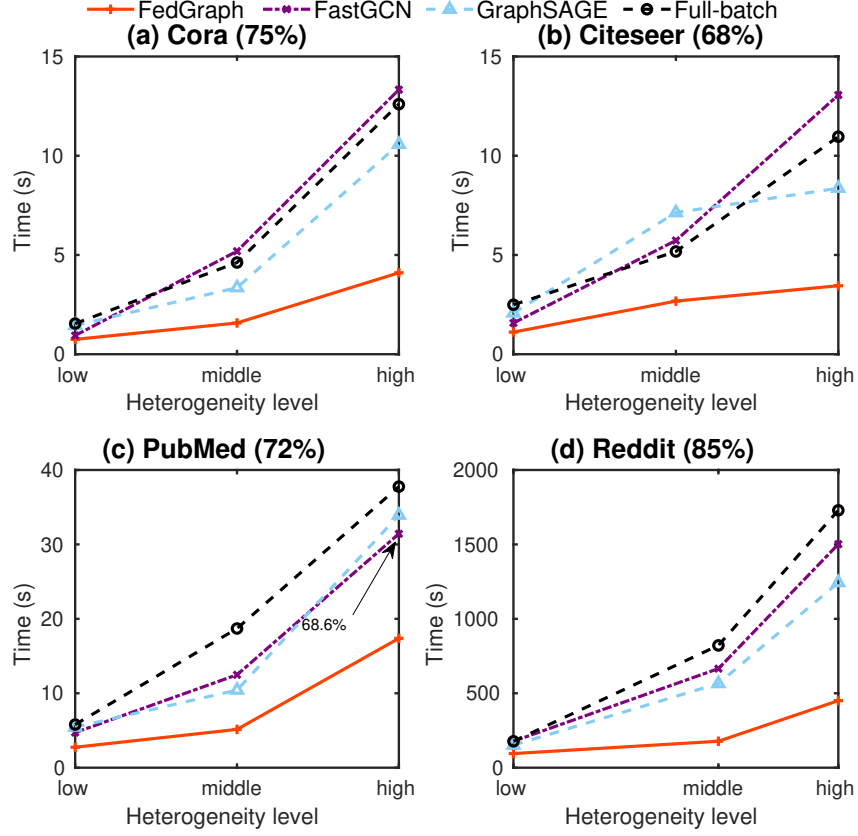


Figure 5.8: The convergence time under different levels of graph heterogeneity.

nections as shown in Table 5.1, and ignoring cross-client edges would seriously break the whole graph structure. Note that FedGraph\_nonShare completes 300-round training earlier because it eliminates embedding sharing.

**The impact of GCN depth.** We study the impact of GCN depth by changing the number of graph convolutional layers. The results are shown in Fig. 5.10. We can see that for all datasets, there is obvious growth of time complexity as we increase the number of layers from 2 to 4. Meanwhile, the accuracy has little changes. In particular, the accuracy of Citeseer decreases as the growth of GCN layers because of the over-smoothing issue [21, 167, 168].

**The impact of non-iid data.** The effectiveness of FedGraph on handling non-IID data is demonstrated in Fig. 5.11. We generate the non-iid data distribution by selecting a subset of node types for each local graph. The experimental results show that FedGraph still outperforms other schemes.

## 5.6 Summary

In this work, we propose FedGraph as a novel federated graph system to enable privacy-preserving distributed GCN learning. Different from traditional federated learning, FedGraph is more challenging because GCN training process involves embedding sharing among clients. To address this challenge, FedGraph uses a novel cross-client graph convolution operation to compress the embeddings before sharing, so that private

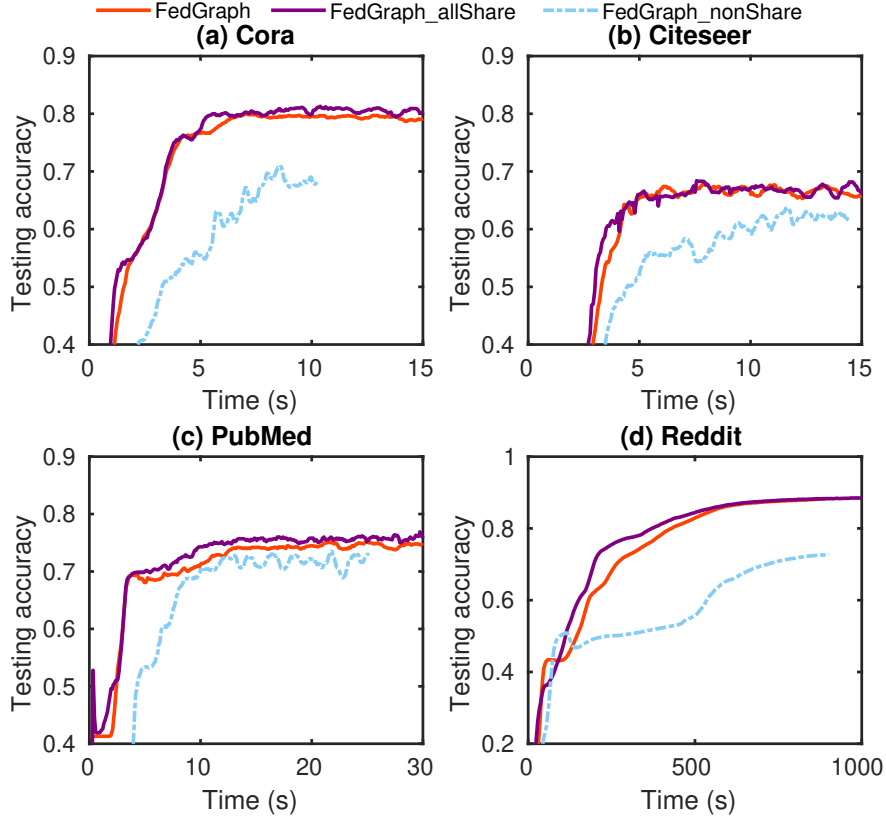


Figure 5.9: Convergence of FedGraph and FedGraph\_nonShare. FedGraph\_nonShare completes the training with less time as it ignores lots of connections in the local training. However, it has a poor convergence.

information can be well hidden. In addition, to reduce GCN training overhead, FedGraph adopts a DRL-based sampling scheme that can well balance the training speed and accuracy. Experimental results on a 20-client testbed show that FedGraph significantly outperforms existing schemes.

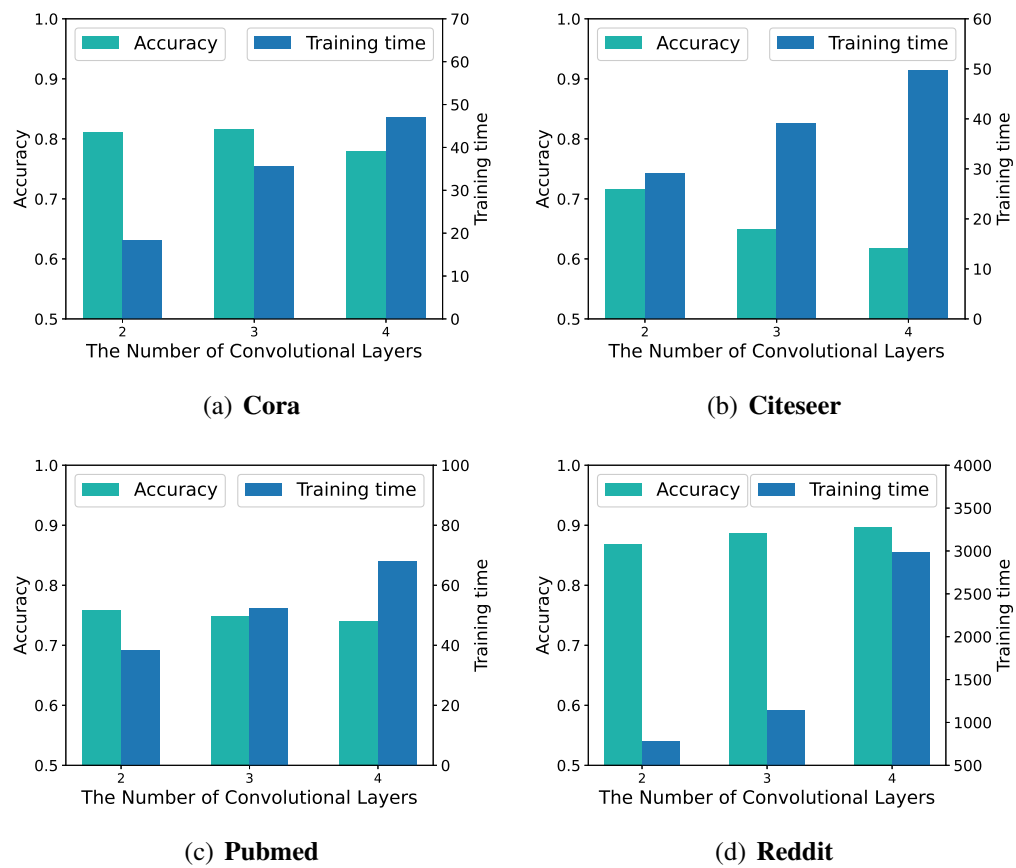


Figure 5.10: Training accuracy and time under different GCN depths.

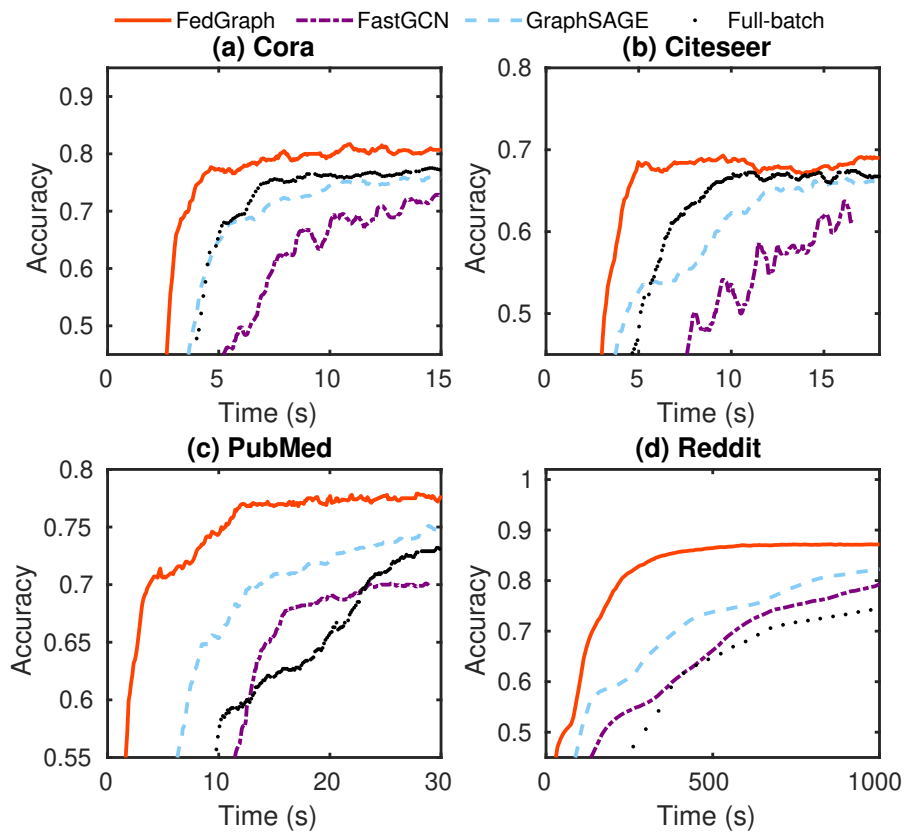


Figure 5.11: Accuracy convergence of different sampling schemes on non-iid data.

## Chapter 6

# Efficient Giant Model Inference on Edges using Graph Learning

### 6.1 Problem Statement

The recent rapid evolution of Artificial Intelligence (AI), notably in deep learning, coupled with advancements in sensor technologies and high-speed communication networks, have significantly stimulated the applications of AI-driven Unmanned Aerial Vehicles (UAVs) in various areas, such as agriculture [169–171] and logistics [172, 173]. Recent achievements in deep learning have demonstrated a positive correlation between model size and accuracy [115, 174]. This insight motivates numerous efforts towards the development of “*Giant Models*”, to enable more complex tasks with sophisticated reasoning prowess on UAVs, such as aerial navigation, manipulation, and embodied agents [175].

However, such tasks rely on giant model inference operations with high computation and memory cost, which cannot be afforded by resource-constrained UAVs. For example, the Switch Transformer model [76], which incorporates 256 experts in each MoE block, requires over 50 GB of memory to load its model weights. In contrast, the available RAM on currently mainstream Unmanned Aerial Vehicles (UAVs) varies between 2 GB and 16 GB. This capacity is extremely insufficient for accommodating models of such sheer size. Existing works present two primary solutions. The first one is to offload inference tasks from UAVs to more powerful cloud or edge servers. For instance, Ateya et al. [176] have proposed a hybrid offloading mechanism for UAVs in order to minimize latency and energy consumption of computationally intensive tasks. Almutairi et al. [177] used an offloading strategy in a multi-UAV environment, focusing on minimizing total service time. Nonetheless, offloading-based methods raise significant privacy concerns due to the necessity of sending data to cloud or edge servers. Although several works [178–180] allow users to offload portions of data or models—thereby preventing their direct exposure of the inference data to the server—malicious entities may still infer the original input data from the shared intermediate results using various techniques, such as model inversion attacks [181, 182]. Furthermore, the process of sending data to and receiving results from servers incurs substantial delays since they need to transmit intermediate results for each piece of inference data and await the final outputs from the server.

The second approach seeks to reduce model size to achieve local inference on UAVs. Wang et al. [183], for example, use model pruning techniques to reduce model sizes for

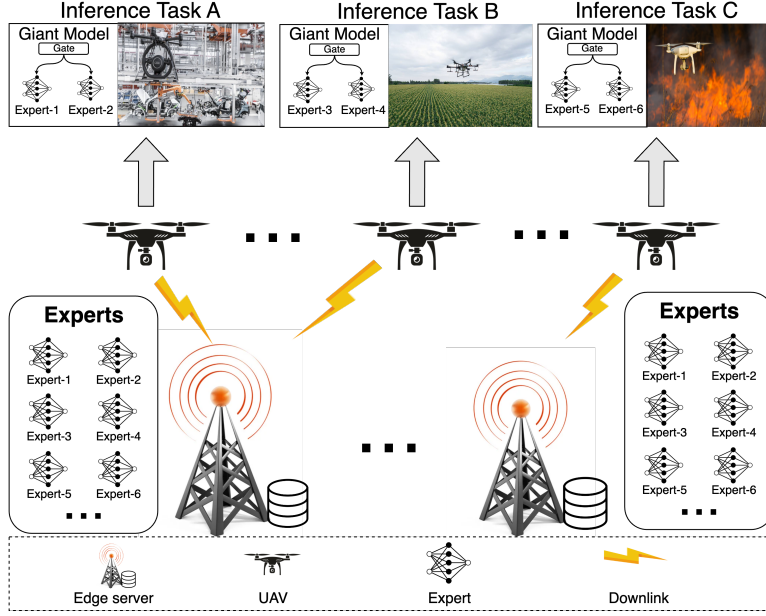


Figure 6.1: The illustration of the proposed inference system of MoE-based models on UAVs. Multiple UAVs download experts from edge servers via wireless networks. The downloaded experts are then used for various inference tasks on UAVs.

efficient UAV tracking. Similarly, the Fault Detection Model Acceleration Engine (FD-MAE) presented in [88] aims to achieve real-time fault detection using a principal component analysis-based model pruning method. While these techniques can effectively reduce model sizes and computational overhead, they often result in lower inference accuracy.

In this work, we propose a new method to enable efficient inference of giant models on resource-constrained UAVs, by exploiting the “mixture-of-experts (MoE)” model architecture. MoE is a typical way to build giant models by decoupling original models into multiple small-scale expert ones, each of which is good at handling a specific kind of input [76, 77]. Although the MoE architecture is effective to reduce computational demands for large models, the model size remains large, posing deployment challenges. The unique characteristics of MoE imply new opportunities of enabling inference serving of giant models on resource-constrained UAVs, without the concerns mentioned above. Specifically, UAVs can dynamically load only a few experts that can well deal with the current input, to reduce the size of memory used for model accommodation. Since UAVs conduct inference operations locally, there is no risk of data exposure to third-party cloud/edge servers and data offloading delay can be also eliminated.

To fully exploit the promises of deploying MoE on UAVs in edge environment, we study a general scenario of several edge servers feeding experts to multiple UAVs, as illustrated in Figure 6.1. All experts are stored in nearby edge servers that are equipped with sufficient resources. During inference, UAVs access edge servers to download their required experts so that they can conduct inference locally. However, achieving an efficient inference serving in the proposed system is challenging. First, UAVs exhibit varied preferences for experts from specific inference data they handle. As shown in our preliminary experiments in Section 6.2, which present the distribution of expert activations for an MoE-based vision model on the ImageNet dataset, different classes

of data activate various experts in each block. However, as mentioned above, UAVs can only load a part of experts due to the limited resources. In addition, there exists a constraint of expert downloading delay to ensure that the online inference tasks will not be responded to with a long delay. Consequently, UAVs need to carefully select experts for downloading, aiming to maximize inference accuracy under resource limitations and the constraint of expert downloading delay. Second, since we consider a multi-UAVs and multi-servers scenario, there exist interactions for the expert downloading between UAVs. Therefore, each UAV also needs to make decisions regarding the association, with the consideration of limited bandwidth and the potential impact of other UAVs' associations, to download more experts under the downloading delay constraint so that the total gained preference can be maximized.

To tackle these challenges, we formulate the expert feeding<sup>1</sup> as an optimization problem. The objective here is to maximize the total preference values obtained by UAVs, while meeting to resource limitations and the constraint of expert downloading delay. The preference value of an expert is related to the activation frequency of this expert for inference data. The higher activation frequency of an expert means more inference data will activate it, and thus the preference value of this expert will be high. This optimization consists of two key decisions: the expert selection and UAV-edge association. Given the non-linear constraints and the large scale of the problem setting, resolving this problem directly within a reasonable time is difficult. Traditional methods typically involve decomposing the problem into two sub-problems and applying specific algorithms to address each iteratively. However, this approach struggles to approximate a globally optimal solution since expert selection and UAV-edge association are tightly coupled. Moreover, with a large-scale problem setting, the convergence of such iterative methods tends to be time-consuming.

In this work, we represent the relationships between UAVs and experts as well as servers as a graph, which provides opportunities for effectively capturing potential interactions between UAVs [184]. We transform the original optimization problem into a graph representation problem. The goal is to find a function that can effectively transform the information of UAVs, experts, and servers into solutions for expert selection and UAV-edge association. To address this representation problem, we introduce GESolver, a method based on graph learning. GESolver integrates a Graph Neural Network (GNN)-based feature extractor with a Multi-Layer Perceptron (MLP)-based solution generator. The GNN component aims to extract hidden features of interactions among UAVs. Subsequently, the MLP-based solution generator processes these extracted features to produce decisions regarding both expert selection and UAV-edge association. The efficacy of GESolver is validated through extensive simulations. Our simulation results indicate that GESolver can significantly outperform other baseline algorithms about  $2.21\times$ .

## 6.2 Motivation

There are several unique characteristics of the MoE architecture, which provide an opportunity to support local inference of giant models on UAVs. First, experts are sparsely activated for specific inference tasks. We study two fine-tuned MoE-based

---

<sup>1</sup>We refer to this as an expert feeding problem for simplicity, considering UAVs require experts from servers.



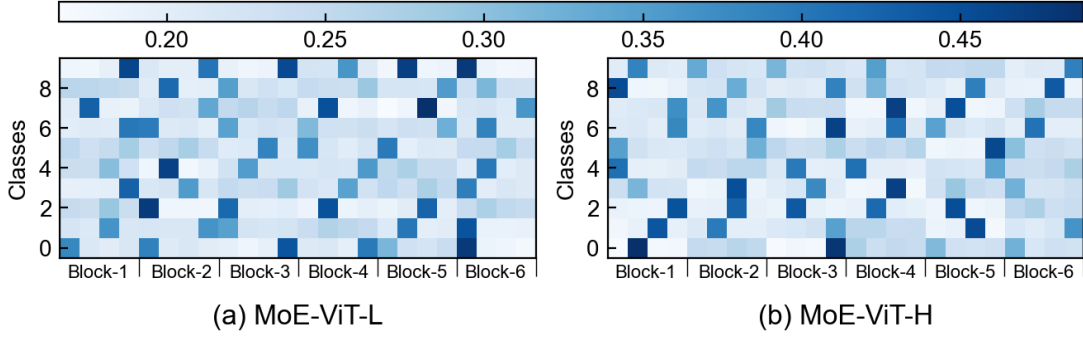


Figure 6.2: Frequencies of expert activation of two MoE-based ViT models on ImageNet dataset. Inference data will sparsely activate different experts.

ViT models [78], MoE-ViT-B and MoE-ViT-L, on an image classification task using the ImageNet dataset [185]. For our experimental purposes, we select 10 classes from this dataset. Both MoE-ViT-B and MoE-ViT-L are configured with 12 blocks, but we focus our analysis on the first 6 blocks, each involving 4 experts. The frequencies of expert activation for these models are depicted in Figure 6.2. This analysis reveals that different inference data prefer different experts. For example, in MoE-ViT-B (Figure 6.2(a)), the majority of class 9 data predominantly activates the fourth expert in the first block. Moreover, as observed in MoE-ViT-L (Figure 6.2(b)), the second expert in the first block is frequently activated by class 0 data, whereas class 1 data shows a preference for the third expert.

Second, experts are skippable. The input and output dimensions of the data processed by the expert are identical. This characteristic implies that the computation within an expert can be skipped without affecting the subsequent inference processes of the entire MoE model. However, as mentioned above, the activation of experts depends on specific inference data. Ignoring experts that will be activated with a high frequency will lead to a significant accuracy drop.

### 6.3 System Overview

The illustration of the proposed system is shown in Figure 6.3. There are two core roles in the system: UAVs and edge servers. The inference serving is managed by a controller, which is deployed in one of the edge servers for fast connection. Periodically, the controller signals all UAVs to load new experts according to their current inference tasks. The operation of the system includes three primary phases:

1. **Profiling.** Each UAV initially profiles its local information<sup>⑩</sup>, which includes its location, preferred experts, the delay constraint for downloads, and others. The UAV's location and memory size can be easily obtained through its status checks. Preferred experts can be estimated using existing prediction methods [186], and the download delay constraint can be determined by checking the current inference workload. This local information from all UAVs is transmitted to the controller via the network module. Simultaneously, the controller collects all relevant data from the edge servers, such as their locations and transmission power. Then, the controller processes this data to produce a global information<sup>⑪</sup>. For instance,

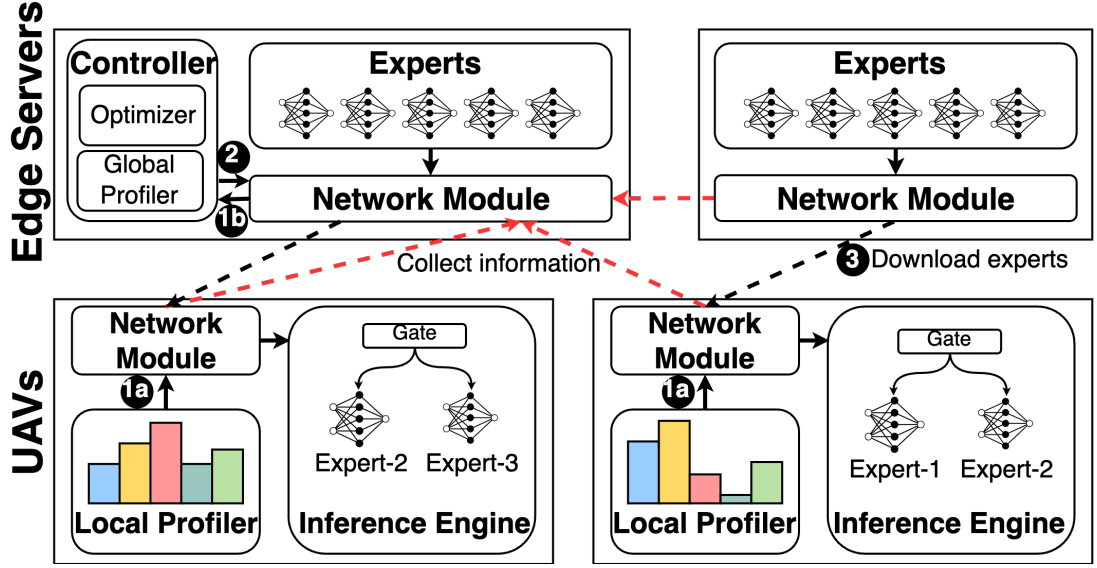


Figure 6.3: An overview of the proposed MoE-based giant model serving system.

the controller needs to calculate the distances between UAVs and edge servers according to their provided locations. After profiling, the necessary data for the following optimization will be ready.

2. **Optimization.** The controller feeds the global information into the optimizer, which then generates the expert feeding decisions<sup>②</sup>, specifically the expert selection and the UAV-edge associations. These expert feeding decisions are subsequently sent to the related UAVs for the following operations.
3. **Inference.** Upon receiving the decisions regarding the expert selection and associations, each UAV first checks which experts should be downloaded from edge servers, according to the local experts from previous downloading. Then, the UAV connects to the designated edge server to download the chosen experts<sup>③</sup>. Once downloaded, the experts will be used to construct an MoE-based model. The model, as well as inference data, are loaded into the UAV's inference engine, starting the inference operations.

The proposed system allows UAVs to perform inference tasks locally, preventing sensitive inference data from being posed to public servers. The efficacy of this system significantly relies on the expert feeding decisions (i.e., expert selection and UAV-edge association). Therefore, the problem lies in making choices about which experts to select and how to associate UAVs with edge servers. In the following sections, we introduce a sophisticated learning-based approach that assists the controller in resolving this issue.

## 6.4 Problem Formulation

We consider a set  $\mathcal{N}$  of UAVs serving a region that is divided into multiple zones. We further divide continuous time into discrete time slots, denoted by the set  $\mathcal{T}$ . In

each time slot, each UAV is assigned one specific zone, i.e., it flies to that zone, collects data, and conducts inference operations [187–189]. Note that each zone is covered by a single UAV and each UAV may be assigned to different zones across different time slots. In this system, we assume that the energy of each UAV is sufficient to support inference operations of a time slot.

At the beginning of each time slot, each UAV downloads experts from edge servers according to inference requests of the target zone. The set of servers is denoted as  $\mathcal{M}$ . The set of available experts is represented by  $\mathcal{E}$ , and each expert  $k \in \mathcal{E}$  has a size of  $s_k$ . We assume that each edge server has sufficient capacity to store all experts. However, UAVs have limited memory capacity and they can only store some experts. We denote the available memory capacity of a specific UAV  $i \in \mathcal{N}$  as  $C_i$ . In addition, the activation of experts relies on the inference input data. Given that UAVs serve different areas and may encounter diverse requests, the preference of specific experts may differ for each UAV. We introduce a value  $f_{i,k}(t)$  to represent the preference of expert  $k$  for UAV  $i$  at time slot  $t$ .  $f_{i,k}(t) \in [0, 1]$  is calculated by the expert activation frequencies:  $f_{i,k}(t) = \frac{a_{i,k}(t)}{\sum_{k' \in \mathcal{E}} a_{i,k'}(t)}$ , where  $a_{i,k}(t)$  is the number of activations of expert  $k$  at time slot  $t$ . The number of activations of experts can be estimated by existing prediction methods [186].

Edge servers co-locate with base stations of wide area networks, which connect to UAVs via wireless networks. Due to the complexity of the environment, we consider both line-of-sight (LoS) and non-line-of-sight (NLoS) links [190, 191] between UAVs and base stations. We use a statistical propagation model to predict the path loss for LoS and NLoS links between UAVs and servers:

$$\eta_{i,j}(t) = d_{i,j}(t)^{-\kappa}, \theta_{i,j}(t) = \varsigma d_{i,j}(t)^{-\kappa}, \quad (6.1)$$

where  $\eta_{i,j}(t)$  and  $\theta_{i,j}(t)$  are LoS and NLoS path loss at time slot  $t$ , respectively. The distance between UAV  $i \in \mathcal{N}$  and server  $j \in \mathcal{M}$  at time slot  $t$  is  $d_{i,j}(t)$ ,  $\kappa$  is the path loss exponent and  $\varsigma$  is the excessive path loss coefficient. We then calculate the probability of path loss at time slot  $t$  according to [192] as:

$$Pr(\eta_{i,j}(t)) = (1 + \alpha \exp(-\beta[\sin^{-1}(\frac{H}{d_{i,j}(t)}) - \alpha]))^{-1}, \quad (6.2)$$

$$Pr(\theta_{i,j}(t)) = 1 - Pr(\eta_{i,j}(t)), \quad (6.3)$$

where  $\alpha$  and  $\beta$  are constants depending on environmental factors, such as suburban, urban, dense urban, high-rise urban, and so on.  $\sin^{-1}(\frac{H}{d_{i,j}(t)})$  is the elevation angle between UAV  $i$  and server  $j$ , where  $H$  is the height of the altitude UAVs. Now, we can calculate the average path loss by [192]:

$$\bar{l}_{i,j}(t) = Pr(\eta_{i,j}(t)) \cdot \eta_{i,j}(t) + Pr(\theta_{i,j}(t)) \cdot \theta_{i,j}(t). \quad (6.4)$$

The signal-to-interference-plus-noise ratio (SINR) of links between UAVs and servers can be calculated as:

$$\gamma_{i,j}(t) = \frac{p_j 10^{-\bar{l}_{i,j}(t)/10}}{\sum_{j' \in \mathcal{U}, j' \neq j} p_{j'} 10^{-\bar{l}_{i,j'}(t)/10} + \delta^2}, \quad (6.5)$$

where  $p_j$  is the transmission power of server  $j$  and  $\delta^2$  is the variance of Gaussian noise.

For UAV  $i$  which is associated with server  $j$ , the data transmission rate can be calculated by:

$$r_{i,j}(t) = \frac{S_i(t)}{\sum_{i' \in \mathcal{N}_j(t)} S_{i'}(t)} B_j \log_2(1 + \gamma_{i,j}(t)), \quad (6.6)$$

where  $S_i(t)$  is the total downloading size for UAV  $i$  and  $\mathcal{N}_j(t)$  are sets of associated UAVs for server  $j$  at time slot  $t$ , respectively. Then, the expert downloading delay for user  $i$  from server  $j$  at time slot  $t$  can be calculated by:

$$Q_{i,j}(t) = \frac{S_i(t)}{r_{i,j}(t)}. \quad (6.7)$$

To ensure timely processing of all inference requests, expert downloading time of each UAV  $i$  at time slot  $t$  is constrained by  $D_i(t)$ . All notations are summarized in Table 6.1.

We aim to maximize the total preference values of experts downloaded by UAVs across time slots, which is denoted by the system utility  $R$ . Maximizing the total preference ensures that UAVs are likely to download experts that will be activated by most of their inference data. Such an objective guarantees that the majority of inference tasks are handled by the appropriate experts. As demonstrated in [186], when inference data is processed by the correct experts, accuracy is at its optimal. Consequently, by optimizing the total preference score, we inherently enhance the overall inference accuracy. To achieve this objective, we need to make two optimization decisions: expert selection and UAV-edge association. On the one hand, due to the limited memory space and downloading delay constraint, each UAV should select a subset of experts for downloading to maximize the total gained preferences. On the other hand, each UAV needs to carefully choose a server for association, with considerations of distance and other associations. Let variable  $x_{i,k}(t) = 1$  indicate that UAV  $i \in \mathcal{N}$  selects expert  $k \in \mathcal{E}$  at time slot  $t$ , otherwise  $x_{i,k}(t) = 0$ . In addition, we use a binary variable  $y_{i,j}(t)$  to denote the association between UAV  $i$  and server  $j \in \mathcal{M}$  at time slot  $t$ . The system utility  $R$  is then formulated as:

$$\max_{x,y} R = \sum_{t \in \mathcal{T}} \sum_{i \in \mathcal{N}} \sum_{k \in \mathcal{E}} x_{i,k}(t) f_{i,k}(t), \quad \text{subject to:} \quad (6.8)$$

$$\sum_{j \in \mathcal{M}} y_{i,j}(t) = 1, \forall i \in \mathcal{N}, t \in \mathcal{T}, \quad (6.9)$$

$$\sum_{k \in \mathcal{E}} x_{i,k}(t) s_k \leq C_i, \forall i \in \mathcal{N}, t \in \mathcal{T}, \quad (6.10)$$

$$\sum_{j \in \mathcal{M}} y_{i,j}(t) Q_{i,j}(t) \leq D_i(t), \forall i \in \mathcal{N}, t \in \mathcal{T}, \quad (6.11)$$

$$S_i(t) = \sum_{k \in \mathcal{E}} [x_{i,k}(t) - x_{i,k}(t-1)]^+ s_k, \forall i \in \mathcal{N}, t \in \mathcal{T} \quad (6.12)$$

$$x_{i,k}(t), y_{i,j}(t) \in \{0, 1\}, \forall i \in \mathcal{N}, j \in \mathcal{M}, k \in \mathcal{E}, t \in \mathcal{T}, \quad (6.13)$$

where Constraint (6.9) ensures that each UAV should connect to one server for expert requiring. Constraint (6.10) represents the limited memory capacity for expert loading. Constraint (6.11) indicates that the expert downloading delay cannot exceed

Table 6.1: Notations

$i, \mathcal{N}$	UAV and the set of UAVs
$j, \mathcal{M}$	Server and the set of servers
$k, \mathcal{E}$	Expert and the set of experts
$t, \mathcal{T}$	Time slot and the set of time slots
$\mathcal{N}_j(t)$	The set of associated UAVs to server $j$ at time slot $t$
$S_i^t$	Size of downloading expert for UAV $i$ at time slot $t$
$Q_i(t)$	The expert download delay of UAV $i$ at time slot $t$
$x_{i,k}(t)$	Binary variable indicates whether UAV $i$ selects expert $k$ at time slot $t$
$y_{i,j}(t)$	Binary variable indicates whether UAV $i$ is associated with server $j$ for expert downloading at time slot $t$
$d_{i,j}(t)$	Distance between UAV $i$ and server $j$ at time slot $t$
$p_j$	The transmission power of server $j$
$B_j$	The available bandwidth of server $j$
$s_k$	The size of expert $k$
$C_i$	Memory capacity of UAV $i$
$D_i(t)$	The constraint of expert download delay of UAV $i$
$\eta_{i,j}(t), \theta_{i,j}(t)$	The path loss for LoS and NLoS links between UAV $i$ and server $j$ at time slot $t$
$Pr(\cdot)$	Probability of path loss
$\bar{l}_{i,j}(t)$	Average path loss between UAV $i$ and server $j$ at time slot $t$
$\gamma_{i,j}$	The SINR of links between UAV $i$ and server $j$
$f_{i,k}(t)$	The preference value of expert $k$ for UAV $i$ at time slot $t$
$r_{i,j}(t)$	The transmission rate between UAV $i$ and server $j$ at time slot $t$
$\mathcal{G}$	The expert feeding graph
$\mathbf{V}, \mathbf{E}$	The sets of vertices and edges in $G$
$\mathbf{x}_v$	The vertex feature vector
$\mathbb{N}_v^l$	The set of $l$ -hop neighbors of vertex $v$
$\mathbf{W}^l$	The weight matrix in the $l$ -th GNN layer

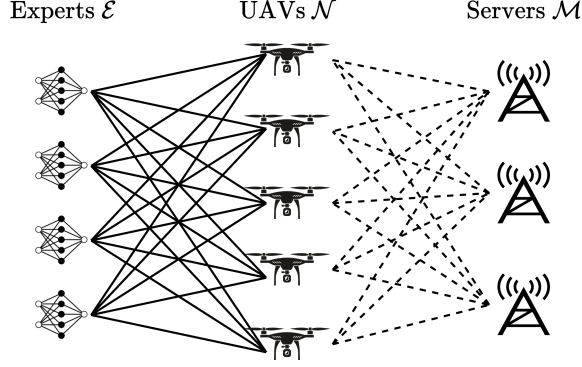


Figure 6.4: The graph to represent the expert feeding problem, where UAVs, servers, and experts are heterogeneous vertices.

the given limitation. Constraint (6.12) represents the downloading size and each UAV only needs to download experts that are not loaded in the previous time slot, where  $[x]^+ = \max\{x, 0\}$ . Finally, Constraint (6.13) gives two binary variables that we need to optimize.

Since we consider a complex multi-UAV and multi-server environment, which exists interactions for expert downloading and edge-association for UAVs, the above optimization problem (6.8) is difficult to solve within a feasible time. While traditional optimization techniques like convex optimization and reinforcement learning could be applied to the expert feeding problem, they often fall short due to the problem's inherent complexity or result in substantial overhead from the long convergence period of algorithms. In this work, we adopt a graph learning-based method to optimize system utility while adding negligible overhead.

## 6.5 Graph Learning-based Expert Feeding

As the relationship between UAVs, experts, and servers can be easily modeled as a graph, we formulate the problem (6.8) (including expert selection and UAV association) as a graph representation learning problem with the objective of finding a representation function to transform the information of UAVs, experts, and servers to solutions of experts selection  $x$ , and UAV association  $y$ . Since it is difficult to find an optimal representation function by hand, we adopt a learning based method to solve this problem. The proposed method for the expert feeding problem is referred to as GESolver. As a whole, GESolver consists of a GNN-based feature extractor and an MLP-based solution generator. We first give the definition of the graph in our system model and then introduce the proposed GESolver in detail.

### 6.5.1 Graph Construction

At each time slot, the expert feeding system can be formulated as a heterogeneous matching graph  $\mathcal{G}$ , involving three different kinds of vertices: UAV vertices, expert vertices, and server vertices, as shown in Fig. 6.4. The links for UAV-expert and UAV-server belong to different types. We generate the vertex feature  $\mathbf{x}_v$  from the information

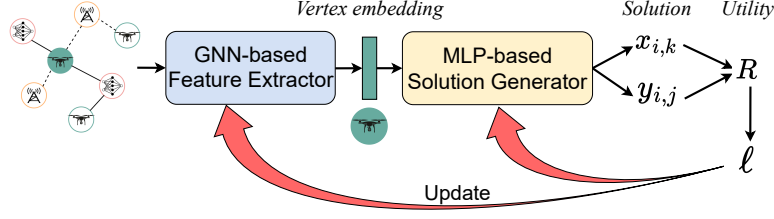


Figure 6.5: The illustration of our proposed graph learning-based method, termed GESolver, which consists of a GNN-based feature extractor and an MLP-based solution generator.

of expert selection and UAV association with different views. Specifically, for the expert vertex, its feature is the frequencies over all UAVs, i.e.,  $\mathbf{x}_v = (f_{1,k}, f_{2,k}, \dots, f_{N,k}) \in [0, 1]^{|N|}$ , where  $\text{type}(v) = \text{'expert'}$  and  $k$  is the corresponding expert for node  $v$ . For the server vertex, its feature consists of SINR of links between this server and all UAVs, i.e.,  $\mathbf{x}_v = (\gamma_{1,j}, \gamma_{2,j}, \dots, \gamma_{N,j}) \in \mathbb{R}^{|N|}$ , where  $\text{type}(v) = \text{'server'}$  and  $j$  is the corresponding server. For a UAV, its feature consists of four parts: the first part is the preferences of all experts and the second part is the selected experts in the previous time slot. The third part is the link SINR for each server and the fourth part is the associated server in the previous time slot. Thus, the feature of a UAV is a vector with a length of  $2 \times |\mathcal{M}| + 2 \times |\mathcal{E}|$ .

### 6.5.2 GESolver Overview

The architecture of GESolver is illustrated in Fig. 6.5, which consists of a GNN-based feature extractor and an MLP-based configuration generator. The input of the GESolver is a heterogeneous graph, which includes heterogeneous vertices of UAVs, experts, and servers, as well as heterogeneous edges. First, the graph is fed into the GNN-based feature extractor to generate vertex embeddings. Then, the generated vertex embeddings are fed into the MLP-based configuration generator to obtain the final solutions, i.e.,  $x$  and  $y$ . We train GESolver in an unsupervised manner. Specifically, to maximize the objective in (6.8), we train GESolver with a loss of:

$$\ell = -R, \quad (6.14)$$

where  $R$  is the optimization objective in (6.8).

We adopt an unsupervised approach to train the GESolver to solve the expert-feeding problem. The training dataset consists of current information about Unmanned UAVs, edge servers, and experts within a time slot. System utility  $R$  serves as the sole guide for model convergence, without the need for any labeled data for vertices or edges in the graph. Furthermore, the learning model undergoes online training during each time slot. The associated training costs are shown in our experiments, which indicates that the process incurs only a negligible overhead to real-time inference.

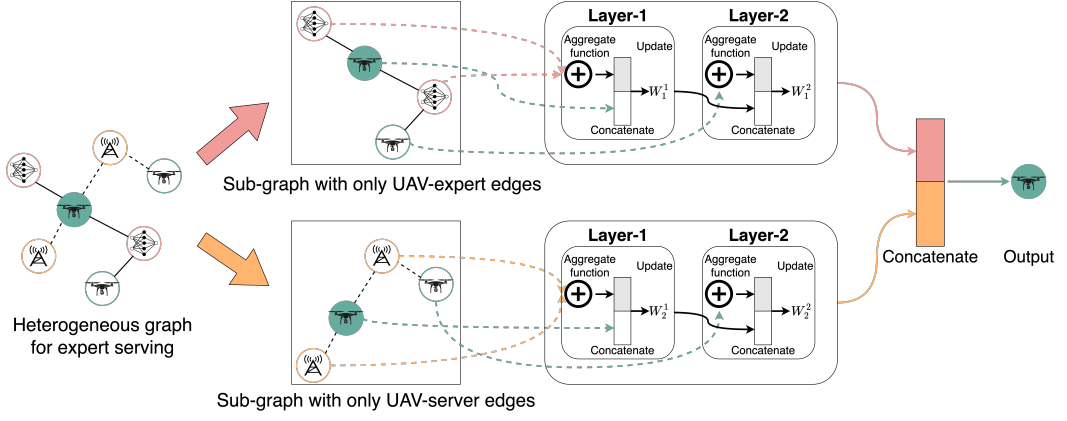


Figure 6.6: The architecture of the GNN-based feature extractor.

**Algorithm 7** GNN-based Feature Extraction Algorithm

---

**Input:** Heterogeneous graph  $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ ;

0: */\* Initialization \*/*

0: Randomly initialize model weights;

0: Divide  $\mathcal{G}$  into two sub-graphs  $\mathcal{G}_1 = (\mathbf{V}_1, \mathbf{E}_1)$  and  $\mathcal{G}_2 = (\mathbf{V}_2, \mathbf{E}_2)$ , according to edge types;

0: **for**  $g = 1, 2$  **do**

0:   **for**  $l = 1, 2, \dots, L$  **do**

0:     **for**  $v \in \mathbf{V}_g$  **do**

0:        $\mathbf{o}_v^{g,l} = \text{AGGREGATE}(\{\mathbf{h}_u^{g,l-1} | u \in \mathbb{N}_v^g\})$

0:        $\mathbf{h}_v^{g,l} = \sigma(\mathbf{W}^{g,l} \cdot \text{CONCAT}(\mathbf{h}_v^{g,l-1}, \mathbf{o}_v^{g,l}))$

0:     **end for**

0:   **end for**

0: **end for**

0:  $\mathbf{h}_v^L = \text{CONCAT}(\mathbf{h}_v^{1,L}, \mathbf{h}_v^{2,L}) = 0$

---

**6.5.3 GNN Based Feature Extractor**

Traditional GNNs, such as GAT [43] and GraphSAGE [122], cannot be directly adopted due to the heterogeneous vertices and edges. To process the heterogeneous edges, we divide the original heterogeneous graph into two sub-graphs, as shown in Fig. 6.6. We then take the GNN operation on each sub-graph. For the heterogeneous vertices, we process them based on an important characteristic of the generated sub-graph. That is, each order of neighbors for a vertex is the same type of vertices. According to this characteristic, we propose to aggregate each order of neighbors and then combine them to generate the final embedding. The details are shown in Algorithm 7. In Algorithm 7,  $\text{CONCAT}(a, b)$  is the concatenation operation of two vectors  $a$  and  $b$ .

**6.5.4 MLP-based Solution Generator**

Since the expert selection and downloading are two couple problems, we thus generate the solutions with the same input via different MLPs, as shown in Fig. 6.7. The vertex embedding is generated from the GNN-based feature extractor. Then, the embed-



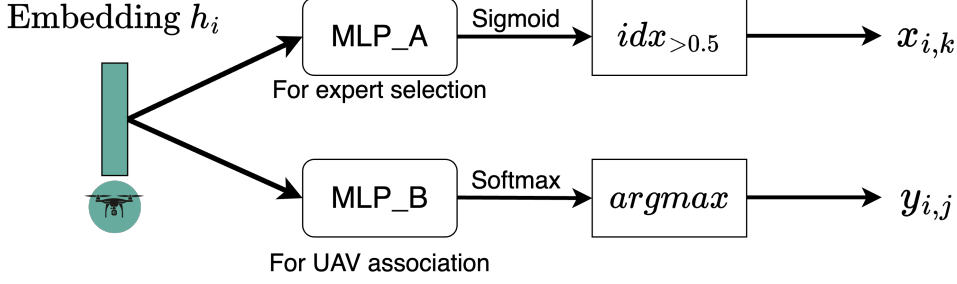


Figure 6.7: The architecture of the MLP-based solution generator.

ding for the user vertex is fed into two different MLPs for generating solutions of expert selection and UAV association. To generate the solution of expert selection, we regard it as a multi-label classification task and use a Sigmoid activation function following the MLP layers. The output from the MLP in the expert selection is  $z_v^s \in [0, 1]^{|E|}$ . We obtain the expert selection solution by:

$$x_{i,k} = \begin{cases} 1, & \text{if } z_v^s[k] > 0.5; \\ 0, & \text{otherwise,} \end{cases} \quad (6.15)$$

where  $i$  is the corresponding user index for vertex  $v$ , and  $k$  is the expert index. We then use a SoftMax activation to generate the UAV association since a UAV can only be associated with one server. Therefore, the output from the UAV association MLP is  $z_v^d \in [0, 1]^{|M|}$ . The final UAV association solution can be obtained by:

$$y_{i,j} = \begin{cases} 1, & \text{if } j = z_v^d; \\ 0, & \text{otherwise,} \end{cases} \quad (6.16)$$

where  $j$  is the UAV index.

### 6.5.5 Algorithm Analysis

Our proposed algorithm is based on the GNN and has a computational complexity of  $\mathcal{O}(L|E|K + L|V|K^2)$ , where  $L$  represents the number of layers in the GNN, and  $K$  denotes the dimensionality of the embedding vectors.  $|V|$  and  $|E|$  indicate the number of vertices and edges within the graph, respectively. In our experimental setup, the largest graph we consider contains 714 vertices and 124,200 edges. This graph includes 300 UAVs, 30 edge servers, and 384 experts, which is a configuration that aligns with those used in existing works [82, 193, 194].

## 6.6 Evaluation

In order to evaluate the performance of our proposed algorithm, GESolver, we perform an extensive set of simulations. The general settings of these simulations are shown in Table 6.2. We compare our proposed GESolver scheme with three baseline techniques: (1) Top-Selection and Maximum SINR Association (TSMA): each UAV in-

Table 6.2: System Parameters

Parameter	Value
<b>General Parameters:</b>	
Path loss constant $\alpha$	11.9
Path loss constant $\beta$	0.13
UAV height $H$	200 m
Expert size $s$	[16, 256] Mbits
Server bandwidth $B_j$	[10, 30] MHz
Transmission power of server $p$	23 dBm
Variance of Gaussian noise $\delta^2$	-174 dBm/Hz
<b>Learning Parameters:</b>	
Learning rate	1e-4
Decay factor	0.99
GNN layers	2
MLP layers	2
GNN hidden dimension	16
MLP hidden dimension	128

iteratively selects experts with the most preference value [195] and associates to a server with the most SINR value [190]; (2) Top-Selection and Learning-based Association (TSLA): in this method, each UAV first selects experts with the most preference value under the memory constraint and then the UAV association is obtained by GESolver, i.e., the MLP-based solution generator only decides the UAV association; (3) Learning-based Selection and Maximum SINR Association (LSMA): each UAV selects experts according to the results of the MLP-based solution generator first and then connects to a server with the most SINR value. The preferences for expert selection are randomly derived from actual MoE model implementations (i.e., MoE-MoE-ViT-L, MoE-ViT-H, and MoE-TransformerXL) and the dataset (i.e., ImageNet and SQuAD [116]).

We study the overall performance, i.e., utility  $R$  defined in (6.8), of our proposed GESolver in Figures 6.8-6.10. First, we evaluate GESolver's performance with different numbers of UAVs, ranging from 100 to 300. We set the number of servers and experts as 10 and 24, respectively. The results are shown in Figure 6.8. We can find that GESolver outperforms the baseline algorithms, achieving performance gain of 85% and 65% over TSLA and LSMA when  $|\mathcal{N}| = 100$ , respectively. Moreover, we can find that the utility  $R$  does not increase linearly with the number of UAVs. The rationale is as follows. With the same number of servers and the constraint for expert downloading delay, the less bandwidth allocated to each UAV, the fewer experts can be downloading. Therefore, although more UAVs contribute to increases in overall obtained utility (i.e., preference), they also increase the associations, resulting in fewer downloaded experts and lower preference per UAV. TSMA algorithms show the worst performance since it greedily select experts and choose servers for UAVs, which cannot capture the potential interactions between UAVs. While TSLA and LSMA can outperform TSMA, thanks to the learning-based solutions, they still fail to match the efficiency of GESolver since they do not jointly consider the expert selection and UAV association.

We then fix the number of UAVs and experts and vary the number of servers from 10 to 20 to evaluate GESolver's performance, as illustrated in Figure 6.9. The numbers of UAVs and experts are set to 100 and 24, respectively. We can find an increase

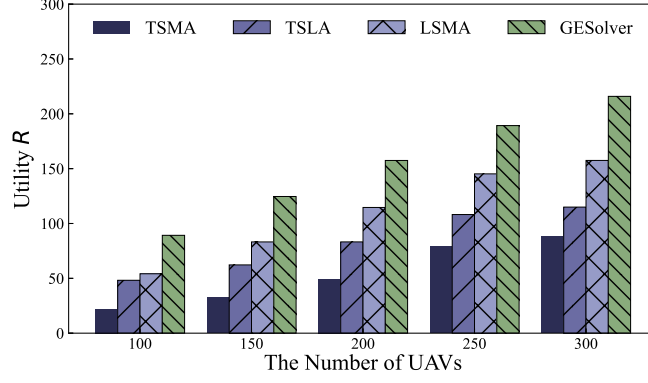


Figure 6.8: Overall performance under different number of UAVs.

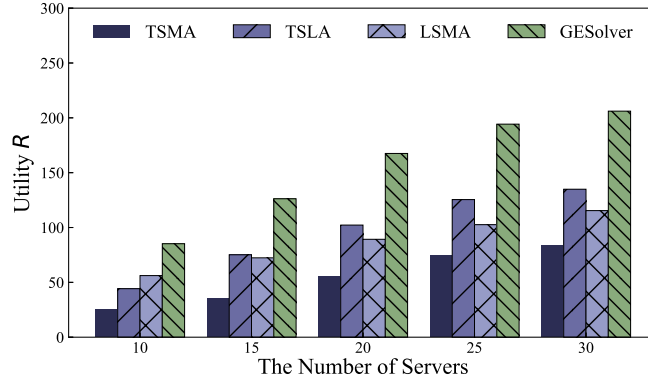


Figure 6.9: Overall performance under different number of servers.

in total utility corresponding to the increasing number of servers. This improvement is attributed to the increased bandwidth availability for UAVs, which increases downloading expert and thereby led to higher utility gains. As a point of reference, when  $|\mathcal{M}|=30$ , TSLA, which learns UAV associations for better bandwidth allocation, outperforms TSMA and LSMA. GESolver has a performance gain of 53% over TSLA, since it optimizes both expert selection and UAV association jointly.

Next, we increase the number of experts from 24 to 384 while maintaining fixed numbers of UAVs and servers, as shown in Figure 6.10. We set the numbers of UAVs and servers as 300 and 10, respectively. We find a decrease in total utility with the increase of experts. The underlying reason for this trend is in two parts. First, a larger pool of experts means each expert is less frequently activated, leading to diminished preference values for UAVs. Consequently, even with the same number of downloaded experts, the aggregate of preference values decreases. Second, the large number of experts results in more variability in expert preference for a given UAV across different time slots. The downloading delay constraint restricts UAVs from acquiring a substantial number of new, highly preferred experts. Our GESolver, designed to learn optimal expert selection with the consideration of downloaded experts in the previous time slot, exhibits a more resilient performance against the increase in the number of experts. Compared to other algorithms, GESolver demonstrates superior performance, outperforming them with a gain of  $1.07\times$  and  $1.35\times$  over TSLA and LSMA when  $|\mathcal{E}| = 384$ ,

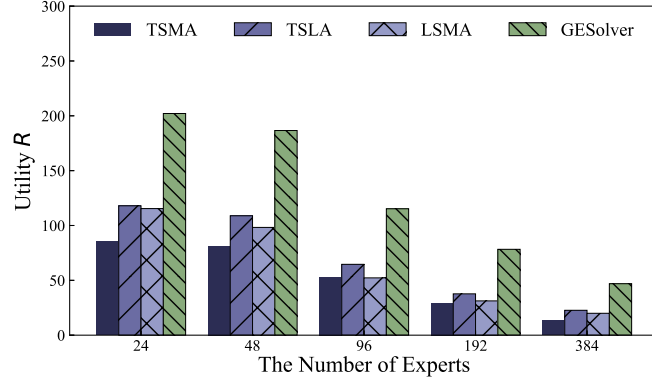


Figure 6.10: Overall performance under different number of experts.

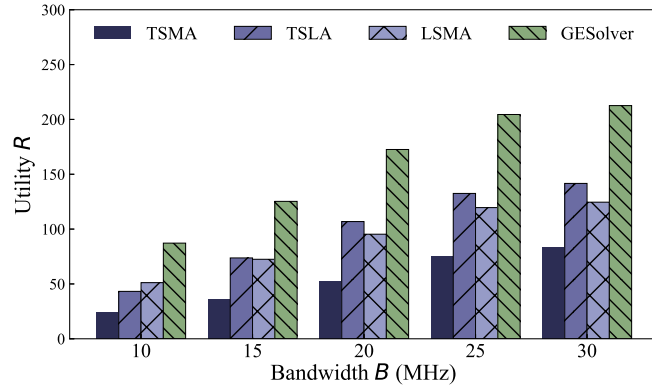


Figure 6.11: Overall performance under different bandwidth.

respectively.

The performance of GESolver under varying bandwidth conditions for UAVs is shown in Figure 6.11. With the increment in bandwidth, we observe an improvement in the total utility for all algorithms. This is because increased bandwidth enables the downloading of a greater number of experts. GESolver, in particular, with a performance gain of  $1.02\times$  and 70% over TSLA and LSMA when  $B = 10\text{MHz}$ , respectively. Furthermore, the total utility experiences marginal gains beyond a certain bandwidth threshold. The rationale behind this is as follows: while more bandwidth allows the downloading of experts with low delay, thus enabling UAVs to download more experts within the downloading delay constraint, the limited memory capacity of each UAV caps the number of experts that can be loaded. Consequently, beyond a certain point, increasing the bandwidth does not proportionally improve the total utility due to this memory limitation.

We next evaluate the performance of GESolver under other different settings, as shown in Figure 6.12. We also change the bandwidth to show the performance of GESolver, but under different combinations of capacity and expert size. Overall, with the increment in bandwidth, we can find an improvement in the total utility for all algorithms, which is because increased bandwidth enables UAVs to download a greater number of experts. However, the total utility also experiences marginal gains beyond a certain bandwidth threshold when we set the capacity as 256 Mbits. The rationale be-

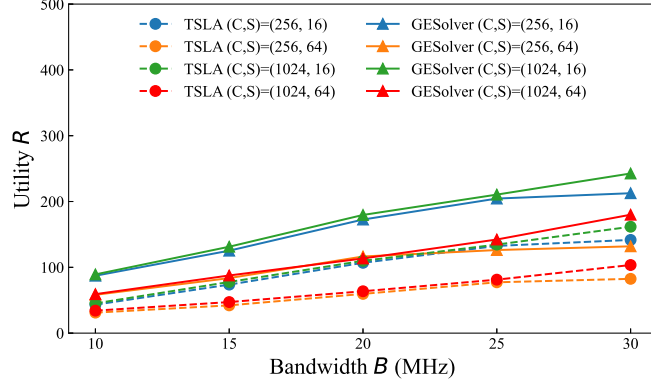


Figure 6.12: Overall performance under different other settings. For example,  $(C, S) = (256, 16)$  indicates a setting of a 256MB of capacity and each expert has a size of 16MB.

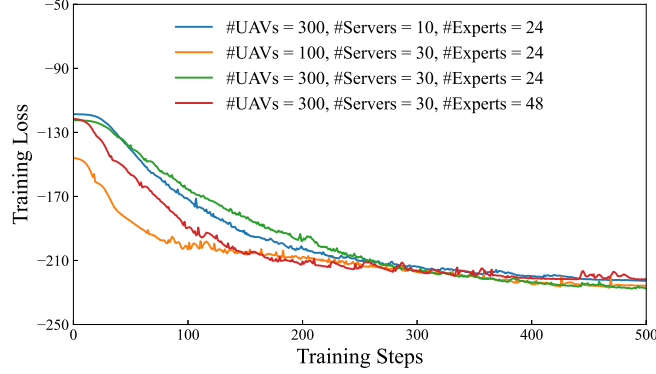


Figure 6.13: Training loss of GESolver.

hind this is as follows: while more bandwidth allows the downloading of experts with low delay, thus enabling UAVs to download more experts within the downloading delay constraint, the limited memory capacity of each UAV caps the number of experts that can be loaded. Consequently, beyond a certain point, increasing the bandwidth does not proportionally improve the total utility due to this memory limitation. Increasing the capacity makes a further improvement of total utility under a larger bandwidth, e.g., a capacity of 1024 Mbits. Furthermore, when we set a large expert size, from 16 Mbits to 64 Mbits, the total utility experiences a significant drop. This is because a larger expert size implies a longer downloading duration per expert and, due to finite memory capacities and bandwidth, restricts the number of experts that can be downloaded. An increment in capacity and bandwidth can enable a greater number of downloading experts, resulting in the improvement of total utility.

We show the convergence of the proposed GESolver under different problem settings in Figure 6.13. The loss value is negative since we aim to maximize the system utility, i.e., minimize  $\ell = -R$ . From the results, we can find that GESolver rapidly converge in around 450 steps, and we can get the solutions of expert selection as well as UAV-edge association. GESolver converges faster under a setting of the number of UAVs as 100, compared to other settings. This is because this setting involves fewer solutions (i.e., expert selection  $x_{i,k}$ ) and thus GESolver can quickly converge. Moreover,

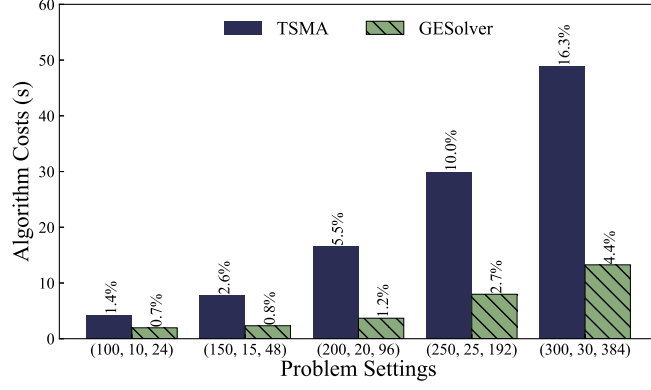


Figure 6.14: The time costs of different algorithms under different problem settings. For example, (100, 10, 24) indicates a setting of 100 UAVs, 10 servers, and 24 experts.

increase the number of UAVs and servers simultaneously (i.e., the third setting) does not bring significant improvement, due to the limited memory size.

Figure 6.14 shows the overhead for TSMA and our proposed GESolver across various problem settings. We report the overhead as a percentage of the total inference time. We conduct the cost comparison to only our method and TSMA. Both TSMA and LSMA employ components from GESolver for expert selection and associating UAVs with edge servers. Consequently, the time costs of TSMA and LSMA align with those of our method. Our comparison with TSMA aims to evaluate the convergence speed, contrasting our proposed learning-based approach with TSMA’s iterative method of solving sub-problems. All costs are measured at an edge server with a CPU of 3.4GHz and 24GB RAM, which is sufficient to load the expert feeding graph in our GESolver. In smaller settings, such as a setting of 100 UAVs, 10 servers, and 24 experts, TSMA incurs negligible time costs, similar to our GESolver. However, as the problem size grows, TSMA’s time to convergence increases substantially. Conversely, our proposed GESolver demonstrates rapid convergence, delivering high-quality solutions in a shorter time, introducing an additional overhead of less than 5%. Moreover, our GESolver can be accelerated with advanced accelerator and framework, e.g., GPUs and PyTorch, while TSMA has no opportunity for this improvement.

## 6.7 Summary

In this work, we propose an inference serving approach for giant models on resource-constrained UAVs, by exploiting the MoE architecture. In this framework, UAVs only load required experts from nearby edge servers to conduct local inference. To ensure an efficient inference serving, we first formulate an expert-serving problem, with the objective of maximizing total gained preference under the constraint of expert download delay and limited resources. To address this problem, we propose a graph learning-based method, termed GESolver. This method is composed of a GNN-based feature extractor and an MLP-based solution generator. Through extensive simulations, we demonstrate that our proposed GESolver method can significantly outperform other existing algorithms.

# Chapter 7

## Conclusion and Future Works

**Conclusion.** The purpose of this dissertation is to comprehensively improve the performance of distributed graph learning. To achieve this objective, this dissertation introduces four novel works by addressing distinct challenges in distributed graph learning.

The first work considers distributed graph learning on multiple heterogeneous GPUs, which poses a significant challenge for model synchronization. In addition, multiple learning jobs often compete for GPU resources. To tackle these challenges, our first work introduces Hare, an efficient job scheduler for distributed graph learning jobs. Hare introduces a novel scheduling algorithm that aims to minimize the total job completion time while maximizing GPU resource utilization.

The second work studies dynamic graphs. Compared to static graphs, processing distributed learning on dynamic graphs introduces more significant communication costs due to both spatial and temporal connections between nodes. To reduce these communication costs, our second work proposes DGC, which includes a novel partitioning method for dynamic graphs. This partitioning method effectively reduces communication costs for distributed dynamic graph learning. Additionally, we propose two runtime optimizations to further enhance the performance of distributed dynamic graph learning.

Our third work focuses on the privacy issues when conducting distributed graph learning across multiple data centers. Specifically, graph data may be distributed among different data centers and cannot be shared due to sensitive information. To enable multiple data centers to collaboratively train the GNN model while preserving privacy, we introduce FedGraph, which includes a federated graph learning algorithm allowing different data centers to share their GNN models instead of the graph data. To accelerate the training process, we also propose a DRL-based intelligent sampling method, which strategically determines the sampling policy for different data centers according to their computation capabilities and graph sizes.

Our final work further enhances the privacy of graph learning by studying machine unlearning technologies. However, existing machine unlearning methods are designed for traditional image data, where data points are independent of each other. For graph data, nodes have connections, and their information can exist in neighboring nodes. To achieve effective unlearning on graph data, we propose a novel push-pull tuning method. This method first modifies the loss function to effectively remove sensitive node information from the graph, referred to as the push-tuning operation. Then, we adopt a pull-tuning operation to retrain the GNN model on the remaining nodes to preserve model accuracy.

---

**Future Works.** Based on the works presented in this dissertation, there are several promising directions that can be explored in the future works.

- **Scalability:** One of the future works can focus on enhancing the scalability and efficiency of the proposed job scheduling. Investigating new algorithms for better resource allocation and job scheduling in even larger and more heterogeneous GPU clusters could further optimize Hare’s performance.
- **Advanced Privacy Techniques:** While FedGraph and the proposed push-pull tuning method address privacy concerns, exploring advanced cryptographic techniques and differential privacy methods could provide even stronger privacy guarantees. However, introducing these technologies will also result in additional computation and communication costs. How to design a privacy-enhanced distributed graph learning method while reducing the training costs is a future work.
- **New Graph Types:** As the variety of graph data continues to grow, developing adaptable methods for new and emerging types of graphs, such as hypergraphs and heterogeneous graphs, could ensure the applicability of distributed graph learning techniques.
- **Integration with Edge Computing:** Investigating the integration of distributed graph learning methods with edge computing paradigms could enable efficient processing of graph data closer to the source, reducing latency and bandwidth usage. Our future works will also explore the distributed graph learning on the edge environment.
- **Applications:** Applying the proposed methods to real-world scenarios, such as social network analysis, recommendation systems, and biological network modeling, could provide valuable insights into their practical effectiveness and limitations. One of our future works aims to deploy our proposed technologies into real applications.



# References

- [1] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti, “Sign: Scalable inception graph neural networks,” *arXiv preprint arXiv:2004.11198*, 2020.
- [2] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, “Lightgcn: Simplifying and powering graph convolution network for recommendation,” in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, 2020, pp. 639–648.
- [3] F. Wu, T. Zhang, A. Holanda de Souza, C. Fifty, T. Yu, and K. Q. Weinberger, “Simplifying graph convolutional networks,” in *Proceedings of Machine Learning Research*, 2019.
- [4] M. Bahri, G. Bahl, and S. Zafeiriou, “Binary graph neural networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 9492–9501.
- [5] Z. Liu, K. Zhou, F. Yang, L. Li, R. Chen, and X. Hu, “Exact: Scalable graph neural networks training via extreme activation compression,” in *International Conference on Learning Representations*, 2021.
- [6] A. Tailor Shyam, F.-M. Javier, and D. Lane Nicholas, “Degree-quant: Quantization-aware training for graph neural networks,” *arXiv preprint arXiv:2008.05000*, 2020.
- [7] H. Wang, D. Lian, Y. Zhang, L. Qin, X. He, Y. Lin, and X. Lin, “Binarized graph neural network,” *World Wide Web*, vol. 24, pp. 825–848, 2021.
- [8] J. Wang, Y. Wang, Z. Yang, L. Yang, and Y. Guo, “Bi-gcn: Binary graph convolutional network,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 1561–1570.
- [9] Y. Wang, B. Feng, and Y. Ding, “Qgtc: accelerating quantized graph neural networks via gpu tensor core,” in *Proceedings of the 27th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2022, pp. 107–119.
- [10] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, “Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks,” in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 257–266.

- [11] Y. You, T. Chen, Z. Wang, and Y. Shen, “L2-gcn: Layer-wise and learned efficient training of graph convolutional networks,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2127–2135.
- [12] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, “Graphsaint: Graph sampling based inductive learning method,” in *International Conference on Learning Representations*, 2019.
- [13] X. Deng and Z. Zhang, “Graph-free knowledge distillation for graph neural networks,” pp. 2321–2327, 2021.
- [14] B. Yan, C. Wang, G. Guo, and Y. Lou, “Tinygcn: Learning efficient graph neural networks,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1848–1856.
- [15] W. Zhang, X. Miao, Y. Shao, J. Jiang, L. Chen, O. Ruas, and B. Cui, “Reliable data distillation on graph convolutional network,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1399–1414.
- [16] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, “Improving the accuracy, scalability, and performance of graph neural networks with roc,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.
- [17] C. Wan, Y. Li, A. Li, N. S. Kim, and Y. Lin, “Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling,” *Proceedings of Machine Learning and Systems*, vol. 4, pp. 673–693, 2022.
- [18] C. Wan, Y. Li, C. R. Wolfe, A. Kyrillidis, N. S. Kim, and Y. Lin, “Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication,” in *International Conference on Learning Representations*, 2021.
- [19] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, “Distdgl: distributed graph neural network training for billion-scale graphs,” in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2020, pp. 36–44.
- [20] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, “Aligraph: a comprehensive graph neural network platform,” *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2094–2105, 2019.
- [21] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv:1609.02907*, 2016.
- [22] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, “Layer-dependent importance sampling for training deep and large graph convolutional networks,” *Advances in neural information processing systems*, vol. 32, 2019.
- [23] F. Chen, P. Li, T. Miyazaki, and C. Wu, “Fedgraph: Federated graph learning with intelligent sampling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1775–1786, 2021.

- 
- [24] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, “Neugraph: Parallel deep neural network computation on large graphs,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 443–458.
  - [25] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, “Dgcl: An efficient communication library for distributed gnn training,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 130–144.
  - [26] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, “Distgmn: Scalable distributed training for large-scale graph neural networks,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.
  - [27] S. Gandhi, A. P. Iyer, H. Xu, T. Rekatsinas, S. Venkataraman, Y. Xie, Y. Ding, K. Vora, R. Netravali, M. Kim *et al.*, “P3: Distributed deep graph learning at scale,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 551–568.
  - [28] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, “Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 515–531.
  - [29] T. Liu, P. Li, and Y. Gu, “Glnt: Decentralized federated graph learning with traffic throttling and flow scheduling,” in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021, pp. 1–10.
  - [30] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, “Bgl: Gpu-efficient gnn training by optimizing graph data i/o and pre-processing,” *arXiv preprint arXiv:2112.08541*, 2021.
  - [31] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, “Neutronstar: distributed gnn training with hybrid dependency management,” in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1301–1315.
  - [32] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou, “Flex-graph: a flexible and efficient distributed framework for gnn training,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 67–82.
  - [33] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim *et al.*, “Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 495–514.
  - [34] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou, “Gnnlab: a factored system for sample-based gnn training over gpus,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 417–434.
-

- [35] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.
- [36] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [37] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, “Agl: A scalable system for industrial-purpose graph machine learning,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12.
- [38] H. Yang, “Aligraph: A comprehensive graph neural network platform,” in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 3165–3166.
- [39] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, “Paragraph: Scaling gnn training on large graphs via computation-aware caching,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 401–415.
- [40] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, “T-gcn: A temporal graph convolutional network for traffic prediction,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 9, pp. 3848–3858, 2019.
- [41] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” in *EMNLP*, 2014.
- [42] A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang, “Dysat: Deep neural representation learning on dynamic graphs via self-attention networks,” in *Proceedings of the 13th international conference on web search and data mining*, 2020, pp. 519–527.
- [43] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” *arXiv preprint arXiv:1710.10903*, 2017.
- [44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [45] B. Yu, H. Yin, and Z. Zhu, “Spatio-temporal graph convolutional networks: a deep learning framework for traffic forecasting,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 3634–3640.
- [46] S. Guo, Y. Lin, H. Wan, X. Li, and G. Cong, “Learning dynamics and heterogeneity of spatial-temporal graph data for traffic forecasting,” *IEEE Transactions on Knowledge and Data Engineering*, 2021.
- [47] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Aste-fanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar, “PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine

- Learning Models,” in *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, 2021, p. 4564–4573.
- [48] H. Zhou, D. Zheng, I. Nisa, V. Ioannidis, X. Song, and G. Karypis, “Tgl: A general framework for temporal gnn training on billion-scale graphs,” *arXiv preprint arXiv:2203.14883*, 2022.
- [49] H. Li and L. Chen, “Cache-based gnn system for dynamic graphs,” in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 937–946.
- [50] X. Song, T. Zhi, Z. Fan, Z. Zhang, X. Zeng, W. Li, X. Hu, Z. Du, Q. Guo, and Y. Chen, “Cambricon-g: A polyvalent energy-efficient accelerator for dynamic graph neural networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 116–128, 2021.
- [51] Y. Wang and C. Mendis, “Tgopt: Redundancy-aware optimizations for temporal graph attention networks,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 354–368.
- [52] C. Wang, D. Sun, and Y. Bai, “Pipad: Pipelined and parallel dynamic gnn training on gpus,” 2023.
- [53] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
- [54] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 17–30.
- [55] I. Stanton and G. Kliot, “Streaming graph partitioning for large distributed graphs,” in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 1222–1230.
- [56] A. Tripathy, K. Yelick, and A. Buluç, “Reducing communication in graph neural network training,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.
- [57] Y. Shao, H. Li, X. Gu, H. Yin, Y. Li, X. Miao, W. Zhang, B. Cui, and L. Chen, “Distributed graph neural network training: A survey,” *ACM Computing Surveys*, vol. 56, no. 8, pp. 1–39, 2024.
- [58] G. Karypis and V. Kumar, “Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices,” 1997.
- [59] M. Guan, A. P. Iyer, and T. Kim, “Dynagraph: dynamic graph neural networks at scale,” in *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2022, pp. 1–10.

- [60] V. T. Chakaravarthy, S. S. Pandian, S. Raje, Y. Sabharwal, T. Suzumura, and S. Ubaru, “Efficient scaling of dynamic graph neural networks,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [61] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, “Spark: Cluster computing with working sets.” *USENIX HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
- [62] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, “Optimus: an efficient dynamic resource scheduler for deep learning clusters,” in *Proceedings of the Thirteenth European Conference on Computer Systems*, 2018, pp. 1–14.
- [63] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, “Themis: Fair and efficient gpu cluster scheduling,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.
- [64] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, “Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.
- [65] Q. Zhang, R. Zhou, C. Wu, L. Jiao, and Z. Li, “Online scheduling of heterogeneous distributed machine learning jobs,” in *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, 2020, pp. 111–120.
- [66] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, “Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
- [67] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, “Heterogeneity-aware cluster scheduling policies for deep learning workloads,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 481–498.
- [68] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu, “Allox: compute allocation in hybrid clusters,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.
- [69] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, “Federated optimization: Distributed machine learning for on-device intelligence,” *arXiv preprint arXiv:1610.02527*, 2016.
- [70] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, “Federated learning with non-iid data,” *arXiv preprint arXiv:1806.00582*, 2018.
- [71] N. H. Tran, W. Bao, A. Zomaya, M. N. H. Nguyen, and C. S. Hong, “Federated learning over wireless networks: Optimization model design and analysis,” in *Proceedings of the International Conference on Computer Communications (INFOCOM)*, 2019, pp. 1387–1395.

- 
- [72] H. Wang, Z. Kaplan, D. Niu, and B. Li, "Optimizing federated learning on non-iid data with reinforcement learning," in *Proceedings of the International Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 1698–1707.
  - [73] T. Suzumura, Y. Zhou, N. Baracaldo, G. Ye, K. Houck, R. Kawahara, A. Anwar, L. L. Stavarache, Y. Watanabe, P. Loyola *et al.*, "Towards federated graph learning for collaborative financial crimes detection," *arXiv preprint arXiv:1909.12946*, 2019.
  - [74] M. Jiang, T. Jung, R. Karl, and T. Zhao, "Federated dynamic gnn with secure aggregation," *arXiv preprint arXiv:2009.07351*, 2020.
  - [75] G. Mei, Z. Guo, S. Liu, and L. Pan, "Sgnn: A graph neural network based federated learning approach by hiding structure," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 2560–2568.
  - [76] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 5232–5270, 2022.
  - [77] C. Riquelme, J. Puigcerver, B. Mustafa, M. Neumann, R. Jenatton, A. Susano Pinto, D. Keysers, and N. Houlsby, "Scaling vision with sparse mixture of experts," *Advances in Neural Information Processing Systems*, vol. 34, pp. 8583–8595, 2021.
  - [78] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale. arxiv 2020," *ICLR*, 2021.
  - [79] S. A. Huda and S. Moh, "Survey on computation offloading in UAV-enabled mobile edge computing," *Journal of Network and Computer Applications*, vol. 201, p. 103341, 2022.
  - [80] J. Lynskey, "Maximizing offloading opportunities for UAV communication," *Korean Netw. Oper. Manage., Jeju Nat. Univ., Jeju, South Korea, Tech. Rep*, 2018.
  - [81] R. Valentino, W.-S. Jung, and Y.-B. Ko, "Opportunistic computational offloading system for clusters of drones," in *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2018, pp. 303–306.
  - [82] Y. Liu, S. Xie, and Y. Zhang, "Cooperative offloading and resource management for UAV-enabled mobile edge computing in power iot system," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 10, pp. 12 229–12 239, 2020.
  - [83] B. Yang, X. Cao, C. Yuen, and L. Qian, "Offloading optimization in edge computing for deep-learning-enabled target tracking by internet of UAVs," *IEEE Internet of Things Journal*, vol. 8, no. 12, pp. 9878–9893, 2020.
  - [84] M. Dai, Z. Su, Q. Xu, and N. Zhang, "Vehicle assisted computing offloading for unmanned aerial vehicles in smart city," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 3, pp. 1932–1944, 2021.
-

- [85] A. Sacco, F. Esposito, G. Marchetto, and P. Montuschi, “A self-learning strategy for task offloading in UAV networks,” *IEEE Transactions on Vehicular Technology*, vol. 71, no. 4, pp. 4301–4311, 2022.
- [86] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, “Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices,” in *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2022, pp. 298–307.
- [87] G. Xu, Z. Hao, Y. Luo, H. Hu, J. An, and S. Mao, “Devit: Decomposing vision transformers for collaborative inference in edge devices,” *IEEE Transactions on Mobile Computing*, 2023.
- [88] B. Wang, X. Peng, M. Jiang, and D. Liu, “Real-time fault detection for UAV based on model acceleration engine,” *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 12, pp. 9505–9516, 2020.
- [89] X. Wang, X. Zhuang, W. Zhang, Y. Chen, and Y. Li, “Lightweight real-time object detection model for UAV platform,” in *2021 International Conference on Computer Communication and Artificial Intelligence (CCAI)*, 2021, pp. 20–24.
- [90] S. Bultmann, J. Quenzel, and S. Behnke, “Real-time multi-modal semantic fusion on unmanned aerial vehicles,” in *2021 European Conference on Mobile Robots (ECMR)*. IEEE, 2021, pp. 1–8.
- [91] M. Vandersteegen, K. Van Beeck, and T. Goedemé, “Super accurate low latency object detection on a surveillance UAV,” in *2019 16th International Conference on Machine Vision Applications (MVA)*. IEEE, 2019, pp. 1–6.
- [92] M. M. Fouda, S. Sakib, Z. M. Fadlullah, N. Nasser, and M. Guizani, “A lightweight hierarchical AI model for UAV-enabled edge computing with forest-fire detection use-case,” *IEEE Network*, vol. 36, no. 6, pp. 38–45, 2022.
- [93] B. Li, Z. Kong, T. Zhang, J. Li, Z. Li, H. Liu, and C. Ding, “Efficient transformer-based large scale language representations using hardware-friendly block structured pruning,” *arXiv preprint arXiv:2009.08065*, 2020.
- [94] M. W. U. Rahman, M. M. Abrar, H. G. Copening, S. Hariri, S. Shao, P. Satam, and S. Salehi, “Quantized transformer language model implementations on edge devices,” *arXiv preprint arXiv:2310.03971*, 2023.
- [95] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.
- [96] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, “Gandiva: Introspective cluster scheduling for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.



- 
- [97] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, “Pipeswitch: Fast pipelined context switching for deep learning applications,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 499–514.
  - [98] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, “Tiresias: A gpu cluster manager for distributed deep learning,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 485–500.
  - [99] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, “Antman: Dynamic scaling on gpu clusters for deep learning,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.
  - [100] “Apache hadoop,” 2021. [Online]. Available: <http://hadoop.apache.org/>.
  - [101] “Cuda Multi-Process Service,” 2019. [Online]. Available: [https://docs.nvidia.com/deploy/pdf/CUDA\\_Multi\\_Process\\_Service\\_Overview.pdf](https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf)
  - [102] P. Yu and M. Chowdhury, “Salus: Fine-grained gpu sharing primitives for deep learning applications,” *arXiv preprint arXiv:1902.04610*, 2019.
  - [103] Y. Yadlapalli, H. Zhou, Y. Zhang, and C. Liu, “gguard: Enabling leakage-resilient memory isolation in gpu-accelerated autonomous embedded systems,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 817–822.
  - [104] H. Fang, M. Doroslovački, and G. Venkataramani, “Eraseme: A defense mechanism against information leakage exploiting gpu memory,” in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 319–322.
  - [105] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, “Superneurons: Dynamic gpu memory management for training deep neural networks,” in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 41–53.
  - [106] M. R. Garey and D. S. Johnson, “Computers and intractability: A guide to the theory of np-completeness,” 1979.
  - [107] M. Queyranne, “Structure of a simple scheduling polyhedron,” *Mathematical Programming*, vol. 58, no. 1, pp. 263–285, 1993.
  - [108] “Technical report,” 2021. [Online]. Available: [https://www.dropbox.com/s/vkdip7xbhh5ca06/Technical\\_report.pdf?dl=0](https://www.dropbox.com/s/vkdip7xbhh5ca06/Technical_report.pdf?dl=0)
  - [109] “grpc,” 2020. [Online]. Available: <https://grpc.io>
  - [110] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR Conference Track Proceedings*, 2015. [Online]. Available: <http://arxiv.org/abs/1409.1556>
-

- [111] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” 2009.
- [112] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [113] D. Martin, C. Fowlkes, D. Tal, and J. Malik, “A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics,” in *Proceedings Eighth IEEE International Conference on Computer Vision (ICCV)*, vol. 2, 2001, pp. 416–423.
- [114] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [115] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *NAACL*, 2019.
- [116] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, “SQuAD: 100,000+ questions for machine comprehension of text,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 2383–2392. [Online]. Available: <https://www.aclweb.org/anthology/D16-1264>
- [117] “Wmt16 dataset,” 2016. [Online]. Available: <http://www.statmt.org/wmt16/>
- [118] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, “Deep speech: Scaling up end-to-end speech recognition,” *arXiv preprint arXiv:1412.5567*, 2014.
- [119] “Common voice dataset,” 2020. [Online]. Available: <https://voice.mozilla.org/>
- [120] J. Chen, T. Ma, and C. Xiao, “Fastgcn: Fast learning with graph convolutional networks via importance sampling,” in *6th International Conference on Learning Representations, ICLR Conference Track Proceedings*, 2018. [Online]. Available: <https://openreview.net/forum?id=rytstxWAW>
- [121] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, “Collective classification in network data,” *AI magazine*, vol. 29, no. 3, pp. 93–93, 2008.
- [122] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” *Advances in neural information processing systems*, vol. 30, 2017.
- [123] “Google Cluster Traces,” 2019. [Online]. Available: <http://github.com/google/cluster-data>
- [124] D. S. Wishart, Y. D. Feunang, A. C. Guo, E. J. Lo, A. Marcu, J. R. Grant, T. Sajed, D. Johnson, C. Li, Z. Sayeeda *et al.*, “Drugbank 5.0: a major update to the drugbank database for 2018,” *Nucleic acids research*, vol. 46, no. D1, pp. D1074–D1082, 2018.

- 
- [125] Y.-C. Lo, S. E. Rensi, W. Torng, and R. B. Altman, “Machine learning in chemoinformatics and drug discovery,” *Drug discovery today*, vol. 23, no. 8, pp. 1538–1546, 2018.
- [126] A. Pal, C. Eksombatchai, Y. Zhou, B. Zhao, C. Rosenberg, and J. Leskovec, “Pinnersage: Multi-modal user embedding framework for recommendations at pinterest,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2311–2320.
- [127] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.
- [128] Y. Wu, D. Lian, Y. Xu, L. Wu, and E. Chen, “Graph convolutional networks with markov random field reasoning for social spammer detection,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 1054–1061.
- [129] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Y. Yeung, “Gaan: Gated attention networks for learning on large and spatiotemporal graphs,” in *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, 2018.
- [130] Y. Ma, Z. Guo, Z. Ren, J. Tang, and D. Yin, “Streaming graph neural networks,” in *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, 2020, pp. 719–728.
- [131] Y. Zhang, S. Gao, J. Pei, and H. Huang, “Improving social network embedding via new second-order continuous graph neural networks,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 2515–2523.
- [132] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, “Evolvegen: Evolving graph convolutional networks for dynamic graphs,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 5363–5370.
- [133] O. A. Malik, S. Ubaru, L. Horesh, M. E. Kilmer, and H. Avron, “Dynamic graph convolutional networks using the tensor m-product,” in *Proceedings of the 2021 SIAM international conference on data mining (SDM)*. SIAM, 2021, pp. 729–737.
- [134] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, “Fennel: Streaming graph partitioning for massive scale graphs,” in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 333–342.
- [135] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, “Graph edge partitioning via neighborhood heuristic,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 605–614.
-

- [136] M. Purohit, B. A. Prakash, C. Kang, Y. Zhang, and V. Subrahmanian, “Fast influence-based coarsening for large networks,” in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1296–1305.
- [137] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis, “Improving graph partitioning for modern graphs and architectures,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015, pp. 1–4.
- [138] G. Bravo Hermsdorff and L. Gunderson, “A unifying framework for spectrum-preserving graph sparsification and coarsening,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [139] Z. Huang, S. Zhang, C. Xi, T. Liu, and M. Zhou, “Scaling up graph neural networks via graph coarsening,” in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 675–684.
- [140] X. Miao, H. Zhang, Y. Shi, X. Nie, Z. Yang, Y. Tao, and B. Cui, “Het: scaling out huge embedding model training via cache-enabled distributed framework,” *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 312–320, 2021.
- [141] J. Peng, Z. Chen, Y. Shao, Y. Shen, L. Chen, and J. Cao, “Sancus: stateless-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks,” *Proceedings of the VLDB Endowment*, vol. 15, no. 9, pp. 1937–1950, 2022.
- [142] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [143] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *Proceedings of the 3rd International Conference for Learning Representations (ICLR’15)*, 2015.
- [144] A. Graves, “Long short-term memory,” *Supervised sequence labelling with recurrent neural networks*, pp. 37–45, 2012.
- [145] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Proceedings of the Artificial Intelligence and Statistics Conference (AISTATS)*, 2017, pp. 1273–1282.
- [146] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, “Federated learning: Strategies for improving communication efficiency,” in *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.
- [147] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, “Graph neural networks: A review of methods and applications,” *arXiv preprint arXiv:1812.08434*, 2018.

- 
- [148] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, “A comprehensive survey on graph neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2020.
  - [149] J. Zhou, C. Chen, L. Zheng, X. Zheng, B. Wu, Z. Liu, and L. Wang, “Privacy-preserving graph neural network for node classification,” *arXiv preprint arXiv:2005.11903*, 2020.
  - [150] C. He, K. Balasubramanian, E. Ceyani, Y. Rong, P. Zhao, J. Huang, M. L. Center, M. Annavaram, and S. Avestimehr, “Fedgraphnn: A federated learning system and benchmark for graph neural networks.”
  - [151] L. Zheng, J. Zhou, C. Chen, B. Wu, L. Wang, and B. Zhang, “Asfgnn: Automated separated-federated graph neural network,” *arXiv preprint arXiv:2011.03248*, 2020.
  - [152] E. Choi, M. T. Bahadori, L. Song, W. F. Stewart, and J. Sun, “Gram: graph-based attention model for healthcare representation learning,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 787–795.
  - [153] S. Scardapane, I. Spinelli, and P. Di Lorenzo, “Distributed graph convolutional networks,” *arXiv preprint arXiv:2007.06281*, 2020.
  - [154] The Top 20 Valuable Facebook Statistics – Updated August 2020, “<https://zephoria.com/top-15-valuable-facebook-statistics>.”
  - [155] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, “Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning,” in *USENIX Annual Technical Conference (ATC 20)*, 2020, pp. 493–506.
  - [156] Y. Aono, T. Hayashi, L. Wang, S. Moriai *et al.*, “Privacy-preserving deep learning via additively homomorphic encryption,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2017.
  - [157] X. Zhang, F. Li, Z. Zhang, Q. Li, C. Wang, and J. Wu, “Enabling execution assurance of federated learning at untrusted participants,” in *Proceedings of the International Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 1877–1886.
  - [158] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, “Occlumency: Privacy-preserving remote deep-learning inference using sgx,” in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–17.
  - [159] J. Chen, J. Zhu, and L. Song, “Stochastic training of graph convolutional networks with variance reduction,” in *Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80*, 2018.
  - [160] W. Huang, T. Zhang, Y. Rong, and J. Huang, “Adaptive sampling towards fast graph representation learning,” in *Advances in neural information processing systems (NeurIPS)*, 2018, pp. 4558–4567.
-

- [161] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, “Practical secure aggregation for privacy-preserving machine learning,” in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.
- [162] R. Shokri and V. Shmatikov, “Privacy-preserving deep learning,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1310–1321.
- [163] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.
- [164] S. Wold, K. Esbensen, and P. Geladi, “Principal component analysis,” *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
- [165] Deep Graph Library, “<https://www.dgl.ai>.”
- [166] scikit-learn, “<https://www.dgl.ai>.”
- [167] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, “Simple and deep graph convolutional networks,” in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, vol. 119, 2020, pp. 1725–1735.
- [168] Y. Rong, W. Huang, T. Xu, and J. Huang, “Dropedge: Towards deep graph convolutional networks on node classification,” in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.
- [169] X. Cao, P. Yang, M. Alzenad, X. Xi, D. Wu, and H. Yanikomeroglu, “Airborne communication networks: A survey,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 9, pp. 1907–1926, 2018.
- [170] Y. Zeng, Q. Wu, and R. Zhang, “Accessing from the sky: A tutorial on UAV communications for 5g and beyond,” *Proceedings of the IEEE*, vol. 107, no. 12, pp. 2327–2375, 2019.
- [171] N. Cheng, S. Wu, X. Wang, Z. Yin, C. Li, W. Chen, and F. Chen, “AI for UAV-assisted iot applications: A comprehensive review,” *IEEE Internet of Things Journal*, 2023.
- [172] M. Patchou, B. Sliwa, and C. Wietfeld, “Unmanned aerial vehicles in logistics: Efficiency gains and communication performance of hybrid combinations of ground and aerial vehicles,” in *2019 IEEE Vehicular Networking Conference (VNC)*. IEEE, 2019, pp. 1–8.
- [173] K. Kuru, D. Ansell, W. Khan, and H. Yetgin, “Analysis and optimization of unmanned aerial vehicle swarms in logistics: An intelligent delivery platform,” *Ieee Access*, vol. 7, pp. 15 804–15 831, 2019.

- 
- [174] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Nee-lakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
  - [175] S. Vemprala, R. Bonatti, A. Buckner, and A. Kapoor, “Chatgpt for robotics: Design principles and model abilities. 2023,” *Published by Microsoft*, 2023.
  - [176] A. A. A. Ateya, A. Muthanna, R. Kirichek, M. Hammoudeh, and A. Koucheryavy, “Energy-and latency-aware hybrid offloading algorithm for UAVs,” *IEEE Access*, vol. 7, pp. 37 587–37 600, 2019.
  - [177] J. Almutairi, M. Aldossary, H. A. Alharbi, B. A. Yosuf, and J. M. Elmirghani, “Delay-optimal task offloading for UAV-enabled edge-cloud computing systems,” *IEEE Access*, vol. 10, pp. 51 575–51 586, 2022.
  - [178] B. Yang, H.-H. Wu, X. Cao, X. Li, T. Kroecker, Z. Han, and L. Qian, “Intelli-eye: An UAV tracking system with optimized machine learning tasks offloading,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2019, pp. 1–6.
  - [179] Z. Zhang, L. L. Njilla, S. Yu, and J. Yuan, “edge-assisted learning for real-time UAV imagery via predictive offloading,” in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–6.
  - [180] P. Zhang, H. Tian, H. Luo, X. Li, and G. Nie, “A hybrid fast inference approach with distributed neural networks for edge computing enabled UAV swarm,” *Physical Communication*, p. 102129, 2023.
  - [181] Z. He, T. Zhang, and R. B. Lee, “Model inversion attacks against collaborative inference,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 148–162.
  - [182] S. Chen, M. Kahla, R. Jia, and G.-J. Qi, “Knowledge-enriched distributional model inversion attacks,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 16 178–16 187.
  - [183] X. Wang, D. Zeng, Q. Zhao, and S. Li, “Rank-based filter pruning for real-time UAV tracking,” in *2022 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 2022, pp. 01–06.
  - [184] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, “Graph neural networks: A review of methods and applications,” *AI open*, vol. 1, pp. 57–81, 2020.
  - [185] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
  - [186] R. Kong, Y. Li, Q. Feng, W. Wang, L. Kong, and Y. Liu, “Serving MoE models on resource-constrained edge devices via dynamic expert swapping,” *arXiv preprint arXiv:2308.15030*, 2023.
-

- [187] Q. Hu, Y. Cai, G. Yu, Z. Qin, M. Zhao, and G. Y. Li, “Joint offloading and trajectory design for UAV-enabled mobile edge computing systems,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1879–1892, 2018.
- [188] T. Zhang, Y. Xu, J. Loo, D. Yang, and L. Xiao, “Joint computation and communication design for UAV-assisted mobile edge computing in iot,” *IEEE Transactions on Industrial Informatics*, vol. 16, no. 8, pp. 5505–5516, 2019.
- [189] M. Yi, X. Wang, J. Liu, Y. Zhang, and B. Bai, “Deep reinforcement learning for fresh data collection in UAV-assisted iot networks,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2020, pp. 716–721.
- [190] C. Chen, T. Zhang, Y. Liu, G. Y. Li, and Z. Zeng, “Joint user association and caching placement for cache-enabled UAVs in cellular networks,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2019, pp. 1–6.
- [191] T. Zhang, Y. Wang, Y. Liu, W. Xu, and A. Nallanathan, “Cache-enabling UAV communications: Network deployment and resource allocation,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 11, pp. 7470–7483, 2020.
- [192] A. Al-Hourani, S. Kandeepan, and S. Lardner, “Optimal lap altitude for maximum coverage,” *IEEE Wireless Communications Letters*, vol. 3, no. 6, pp. 569–572, 2014.
- [193] Y. Guo, S. Gu, Q. Zhang, N. Zhang, and W. Xiang, “A coded distributed computing framework for task offloading from multi-UAV to edge servers,” in *2021 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2021, pp. 1–6.
- [194] Z. Kuang, H. Wang, J. Li, and F. Hou, “Utility-aware UAV deployment and task offloading in multi-UAV edge computing networks,” *IEEE Internet of Things Journal*, 2023.
- [195] Z. Lin and W. Chen, “Content pushing over multiuser MISO downlinks with multicast beamforming and recommendation: A cross-layer approach,” *IEEE Transactions on Communications*, vol. 67, no. 10, pp. 7263–7276, 2019.