EosDNN: An Efficient Offloading Scheme for DNN Inference Acceleration in Local-Edge-Cloud Collaborative Environments

Min Xue, Huaming Wu, *Member, IEEE*, Ruidong Li, *Senior Member, IEEE*, Minxian Xu, *Member, IEEE*, and Pengfei Jiao

***Abstract*—With the popularity of mobile devices, intelligent applications, e.g., face recognition, intelligent voice assistant, and gesture recognition, have been widely used in our daily lives. However, due to the lack of computing capacities, it is difficult for mobile devices to support complex Deep Neural Network (DNN) inference. To alleviate the pressure on these devices, traditional methods usually upload part of the DNN model to a cloud server and perform a DNN query after uploading an entire DNN model. To achieve real-time DNN query, we consider the collaboration between local, edge and cloud, and perform DNN query when uploading DNN partitions. In this paper, we propose an Efficient offloading scheme for DNN Inference Acceleration (EosDNN) in a local-edge-cloud collaborative environment, where the DNN inference acceleration is mainly embodied in the optimization of migration delay and realization of real-time DNN query. EosDNN comprehensively considers the migration plan and uploading plan, where for the former, a Particle Swarm Optimization with Genetic Algorithm (PSO-GA) is applied to obtain the distribution of DNN layers under the server with the lowest migration delay, and for the latter, a Layer Merge Uploading Algorithm (LMU) is proposed to obtain DNN partitions and their upload order with efficient DNN query performance. Experimental results demonstrate that EosDNN can be applied to large-scale DNN model migration, which can achieve an ideal migration delay and obtain a more fine-grained DNN partition uploading plan, thereby optimizing DNN query performance.**

***Index Terms*—Mobile computing, local-edge-cloud collaboration, computation offloading, DNN inference, intelligent applications.**

Manuscript received April 22, 2021; revised June 29, 2021; accepted September 7, 2021. Date of publication September 10, 2021; date of current version February 16, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61801325, Grant 62071327, and Grant 62102408; in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 19H04105; in part by the Shenzhen Institutes of Advanced Technology (SIAT) Innovation Program for Excellent Young Researchers; and in part by the Tianjin Research Innovation Project for Postgraduate Students (Artificial Intelligence) under Grant 2020YJSZXS27. *(Corresponding author: Huaming Wu.)*

Min Xue and Huaming Wu are with the Center for Applied Mathematics, Tianjin University, Tianjin 300072, China (e-mail: xm_17@tju.edu.cn; whming@tju.edu.cn).

Ruidong Li is with the Institute of Science and Engineering, Kanazawa University, Kanazawa 920-1192, Japan (e-mail: liruidong@ieee.org).

Minxian Xu is with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China (e-mail: mx.xu@siat.ac.cn).

Pengfei Jiao is with the Center of Biosafety Research and Strategy, Tianjin University, Tianjin 300072, China (e-mail: pjiao@tju.edu.cn).

Digital Object Identifier 10.1109/TGCN.2021.3111731

I. Introduction

DEEP Neural Networks (DNNs) have already demonstrated overwhelming advantages in computer vision, natural language processing, biological information, driverless cars, and other fields. Along with the large-scale popularization of mobile devices and driven by artificial intelligence technology, more and more intelligent applications based on DNN models have emerged, and the number of DNN intelligent applications deployed on mobile devices has increased dramatically [1]. However, due to the challenges of insufficient computing power and small storage space in mobile devices, it is difficult for them to support the complex operations of DNN inference [2], e.g., the calculation of network weights determined by DNN training. The process of DNN inference by running the DNN layer under the cloud/edge server and the DNN layer under the client is called *DNN query*.

To run complex DNNs on the cloud/edge server, we need to install the corresponding pre-trained DNN model on the target cloud/edge server [3], [4]. Considering the real-time data processing, it is generally preferred to upload the DNN model to a cloud/edge server closer to the task [5], [6]. However, since the client can keep moving, it is quite difficult to accurately predict which cloud/edge server the client will connect to. Thus it is unreasonable to pre-install the DNN model on the cloud/edge server. Therefore, we first need to determine which cloud/edge server the client is connected to, and then deploy the DNN model from the client to the specified cloud/edge server, which is a more realistic approach.

As a promising method, the DNN migration plan can reduce the pressure on mobile devices by migrating extremely computation-intensive execution from resource-constrained devices to cloud/edge servers, thereby achieving DNN inference acceleration [7], [8]. It mainly involves DNN deployment, migrating decision-making, and resource allocation [9]. The traditional method usually migrates part of the DNN model to cloud servers with high computing power, thus reducing the execution delay of the DNN model and achieving low delay DNN migration [10]. However, since the cloud center is too far away from the client, *cloud-based migration plans* are affected by multiple factors, e.g., the network bandwidth, the central computing capacity, the amount of data transferred, and the number of computing tasks [6], [11], [12].

Instead, we can seek the help of edge computing, where edge servers are widely distributed between mobile devices and cloud computing centers, and integrate core capabilities of the network, computing, storage, and applications [13], [14]. Thus, *edge-based migration plans* can effectively alleviate the burden on network bandwidth and achieve lower transmission delay, thereby obtaining a DNN migration plan with low migration delay [15]. Unfortunately, the computing power of edge servers is generally limited, and it is difficult to support the operation of large-scale DNN models. Therefore, capturing a DNN migration plan in a resource-limited computing environment is one of the current challenges. We know that the local-edge-cloud collaborative environment combines the characteristics of cloud computing with high computing capacity and edge computing with low transmission delay. However, most current studies fail to consider how to obtain the DNN migration plan suitable for large-scale DNN models in the multi-user local-edge-cloud collaborative environment with limited computing resources [16], [17].

One of the major problems with DNN uploads is that the transmission delay for uploading the entire DNN model is too long. Before the completion of uploading the DNN model, the DNN query will only be run by the client, which causes a great burden on the client while the DNN query performance is very poor [10], [18]. Therefore, we should consider segmenting the DNN model to obtain the *DNN partition* and execute the DNN query while uploading the DNN partition, thereby increasing DNN query performance. For the DNN partition that has been successfully uploaded to the cloud/edge server, the message that the DNN partition was successfully uploaded will be returned to the client. At this time, the DNN query will be executed by the DNN partition under the client and the DNN partition successfully uploaded to the cloud/edge server. The shortest path method combined with the penalty factor method is often used to make the uploading plan of the DNN model. As a result, DNN partitions are generally large and lack the opportunity to achieve better DNN query performance with more fine-grained partitions. In response to the problem of excessive granularity partitioning, an efficiency-based partitioning algorithm was designed to generate a more fine-grained uploading plan [19]. Unfortunately, similar to the above uploading plans, this method uploads DNN partitions from a single client to a single cloud/edge server, and it is challenging to be further applied to a multi-cloud/edge server environment with multiples users and servers.

To address the above challenges, we propose a novel offloading scheme called *EosDNN*, considering the migration and uploading problems of DNN models in the multi-user local-cloud-edge collaborative computing environment [20]. Firstly, we consider the migration problem of DNN models in the local-edge-cloud collaborative environment with limited resources. In order to obtain the optimal migration delay under multi-task parallelism, we apply the Particle Swarm Optimization with Genetic algorithm (*PSO-GA algorithm*) to create the DNN migration plan. After that, we also consider the processing of the topological DNN model containing the inception module. In order to optimize the migration delay of the DNN model, we propose a *two-step migration* method to process the topological DNN model. Secondly, we consider a DNN uploading plan that allows the client to perform DNN queries before uploading the entire DNN model. With the aim of performing DNN queries more efficiently, we propose a layer merge uploading algorithm (*LMU algorithm*) to create the DNN uploading plan. Compared with existing partition uploading plans, the *LMU algorithm* can generate a more fine-grained DNN partition uploading plan by combining the DNN layer with adjacent layers to form a DNN partition, where the DNN layer is the finest-grained DNN partition. This approach avoids the problem that the granularity of the DNN partition is too large due to the direct partition of the whole DNN model. The main **contributions** of this paper are listed as follows:

- A migration plan based on *PSO-GA algorithm* is proposed to obtain the distribution of the DNN layer under each server in the local-edge-cloud collaborative environment. This method is suitable for multi-task parallelism, which is conducive to obtaining the lowest migration delay. In addition, we propose a *two-step migration* method to deal with topological DNN models.
- An uploading plan based on *LMU algorithm* is proposed to obtain DNN partitions and determine the upload order of DNN partitions. This method greatly reduces the granularity of DNN partitions and greatly improves DNN query performance.
- Consider a more realistic computing environment and offloading scheme, where multiple DNN models are paralleled in a multi-user local-edge-cloud collaborative environment with limited resources, which is suitable for large-scale DNN model migration and upload.

The rest of this paper is organized as follows. Section II discusses relevant studies. Section III introduces a framework of local-edge-cloud collaborative computing. Section IV presents the overall architecture of the *EosDNN* offloading scheme. Section V describes the first part of the *EosDNN* offloading scheme, i.e., the migration plan, and gives the distribution of the DNN layer under the server in the local-edge-cloud collaborative computing. Section VI proposes the second part of the *EosDNN* offloading scheme, i.e., the uploading plan, and formulates a plan for the upload order of the DNN partition. Section VII provides the simulation results from different aspects. Finally, Section VIII concludes the paper and points out future research directions.

II. Related Work

In recent years, many efforts have been devoted to the DNN migration plans and DNN uploading plans, respectively, and several feasible solutions have been put forward from different perspectives.

A. Existing Migration Plans

Previous migration plans are mostly reflected in the application of task migration. Liu *et al.* [21] formulated the optimization problem for energy saving on mobile devices whose tasks can be divided, and utilized a greedy choice to solve the problem. Zhang *et al.* [22] proposed a

lightweight energy-efficient computational migration plan to make migration decisions for each component and adopted a greedy heuristic [21] to determine which components to be migrated to edge servers. Cui *et al.* [23] considered a tripartite-based model to represent the problem of data replica placement. For this reason, they proposed a data replica placement strategy based on the Genetic Algorithm (GA) to reduce data transmission in the cloud server. Pandey *et al.* [24] presented a heuristic algorithm based on Particle Swarm Optimization (PSO) [25] for data-intensive applications while considering both the computational cost and data transmission cost. Deng *et al.* [26] presented a fine-granularity migration plan and the energy-efficient task migration problem is mathematically formulated as a constrained 0-1 programming. Lin *et al.* [27] proposed a self-adaptive Discrete Particle Swarm Optimization algorithm with GA operators (GA-DPSO) [28], [29], optimizing the data transmission delay when placing data for a scientific workflow. This approach considers the impact factors that affect transmission delay and the data placement characteristics. Lin *et al.* [30] proposed an adaptive discrete PSO algorithm using GA operators to reduce the system cost caused by data transmission and DNN model execution.

B. Existing Uploading Plans

Some research has been conducted on DNN uploading with the use of partition-based methods. Jeong *et al.* [31] proposed a new approach to run a Machine Learning (ML) Web app on resource-constrained embedded devices by uploading ML computations to servers, where uploading computations dynamically depending on the problem size and network status. IONN [3] uses the shortest path method and the penalty factor method to determine DNN partitions, and builds the DNN model incrementally when each DNN partition arrives, allowing the client to start partial uploading even before uploading the entire DNN model, thereby improving query performance. Enhanced Partitioning [32] is based on a penalty factor method of uploading overhead and uses the shortest path method on the DNN execution graph between the client and the cloud/edge server to partition the DNN layer, which generates a more granular uploading plan. JointDNN [33] transforms the optimal computing scheduling problem of DNN into the shortest path problem and Integer Linear Programming (ILP) in the mobile cloud computing environment, and divides the DNN architecture by optimization formulations at layer granularity, thereby achieving collaborative computing between mobile devices and the cloud. In addition, Jeong [34] constructed a lightweight edge computing system called PerDNN, to provide seamless uploading services even if users move between multiple edge servers, where PerDNN was proposed to further support DNN to perform uploading between mobile users and many interconnected edge servers.

C. Qualitative Comparison

We briefly analyzed the issues existing in previous migration plans and uploading plans, respectively. Then, in view of the deficiencies of the existing work, we put forward our improvement.

Migration Plan: Swarm intelligence algorithms, e.g., GA and PSO algorithms, are usually considered for the migration plan. However, these algorithms are easy to fall into the local optimum, and it is difficult to obtain global optimal results. Moreover, they neglect the introduction of crossover and mutation operations in the PSO algorithm to optimize the update process. In addition, the existing work only considers how to obtain the optimal migration plan of the DNN model in the edge/cloud environment, and ignores the parallel requirements of large-scale DNN models for multiple users, in which the computing resources of the edge server are insufficient to support DNN operation and the high transmission delay of cloud servers is not enough to achieve low migration delay. In other words, most existing migration plans do not take into account the waiting time caused by the parallelism of large-scale DNN models in resource-constrained computing environments.

Uploading Plan: Most of the current studies are based on the shortest path method and the penalty factor method to formulate the uploading plan of the DNN model. However, the problem of the penalty factor method is that for each DNN model, due to the limited number of penalty factor values, DNN partitions are generally large and lack the opportunity to achieve better DNN query performance. For the problem of excessive granularity, some scholars have proposed efficiency-based partitioning algorithms, which can generate more fine-grained uploading plans. Unfortunately, this uploading plan is only suitable for environments from a single client to a single edge server, not for complex environments with multi-cloud/edge servers. Several lightweight edge computing systems have been developed to provide seamless upload services, but the loss caused by the layer upload delay has been ignored.

As a comparison, this paper proposes an efficient *EosDNN* offloading scheme to speed up DNN inference. It is achieved by considering both the DNN migration plan and the DNN uploading plan. For the migration plan, we propose a *PSO-GA algorithm* in a multi-user local-edge-cloud collaborative environment, which can realize the multi-task parallelism with low migration delay, obtain the distribution of DNN layer under the server, and is suitable for large-scale DNN model migration. For *PSO-GA algorithm*, the crossover operation and mutation operation of *GA algorithm* are introduced into the update of *PSO algorithm*, which is beneficial to obtain the global optimal migration plan. For DNN uploading plans, we propose the *LMU algorithm* that improves DNN query performance by reducing the granularity of the DNN partition, suitable for DNN queries while uploading DNN partitions.

III. Environmental Definition

In this paper, we consider a multi-user local-edge-cloud collaborative computing environment ($\mathbb{M}$, $\mathbb{E}$, $\mathbb{C}$), where $\mathbb{M} = \{m_1, m_2, \ldots, m_u\}$ is the set of clients, $\mathbb{E} = \{e_1, e_2, \ldots, e_o\}$ is the set of edge servers and $\mathbb{C} = \{c_1, c_2, \ldots, c_w\}$ is the set of cloud servers. Here, we set up a local-edge-cloud collaborative environment with u clients, o edge servers, and

TABLE I
SYMBOLS AND DEFINITIONS

Symbol	Definition
$\mathbb{M} = \{m_1, \cdots, m_u\}$	Clients
$\mathbb{E} = \{e_1, \cdots, e_o\}$	Edge servers
$\mathbb{C} = \{c_1, \cdots, c_w\}$	Cloud servers
$\mathbb{R} = \{r_1, \cdots, r_s\}$	Local-edge-cloud collaborative environment
$\mathbb{D}_i = \{D_{i1}, \cdots, D_{ij}, \cdots\}$	The number of DNN layers
D_{ij}	The j-th layer under the i-th DNN
T_{ij}^1	Execution delay
T_{ij}^2	Transmission delay
T_{ij}^3	Waiting delay

w cloud servers. The symbols and definitions are described in Table I.

The multi-user local-edge-cloud collaborative computing environment is defined as $\mathbb{R} = (\mathbb{M}, \mathbb{E}, \mathbb{C}) = \{r_1, r_2, \ldots, r_d, \ldots, r_s\}$, where $\{r_1, r_2, \ldots, r_u\}$ represents the set of clients $\mathbb{M}$, $\{r_{(u+1)}, r_{(u+2)}, \ldots, r_{(u+o)}\}$ represents the set of edge servers $\mathbb{E}$, and $\{r_{(u+o+1)}, r_{(u+o+2)}, \ldots, r_s\}$ represents the set of cloud servers $\mathbb{C}$, respectively. Here, we have $d \in [0, s]$ and $s = u + o + w$. The DNN set on the device is denoted by $\mathbb{D} = \{D_1, D_2, \ldots, D_i, \ldots, \}, i \in [1, \alpha]$, where we define the DNN model D_i as a task, and α indicates the number of tasks. Each DNN task can be expressed as $D_i = \{D_{i1}, D_{i2}, \ldots, D_{ij}, \ldots, \}, j \in [1, \beta_i]$, where D_{ij} refers to the subtask at the j-th layer under the i-th DNN model, and β_i refers to the number of subtasks under the task D_i. In addition, we know that subtasks D_{ij} under each task D_i run in serial order.

IV. ARCHITECTURE OF EOSDNN OFFLOADING SCHEME

We cannot accurately predict which cloud/edge server the mobile client will connect to, so it would not make sense to pre-install DNN subtasks on the cloud/edge server. It is a more realistic method to first determine which cloud/edge server the client will connect to, and then deploy the DNN subtasks under the client to the designated cloud/edge server. In the above scenario, how to achieve efficient DNN inference becomes a problem. To address the above issues, the *EosDNN* offloading scheme is established in a multi-user local-edge-cloud collaborative environment with limited resources. As shown in Fig. 1, we describe the operation process of the *EosDNN* offloading scheme, and then describe the internal architecture of the **migration plan** and an **uploading plan**, where the *EosDNN* offloading scheme contains a **migration plan** and an **uploading plan**, which achieve low migration latency and efficient DNN query performance, respectively.

A. Migration Plan

We consider using *PSO-GA algorithm* to generate a DNN migration plan, so as to determine the distribution of DNN subtasks under the cloud/edge server.

Considering the high transmission delay of cloud servers and the limited computing capacity of edge servers, the existing local-cloud or local-edge computing environment is difficult to support large-scale DNN model inference. In addition, most of the existing algorithms are not suitable for

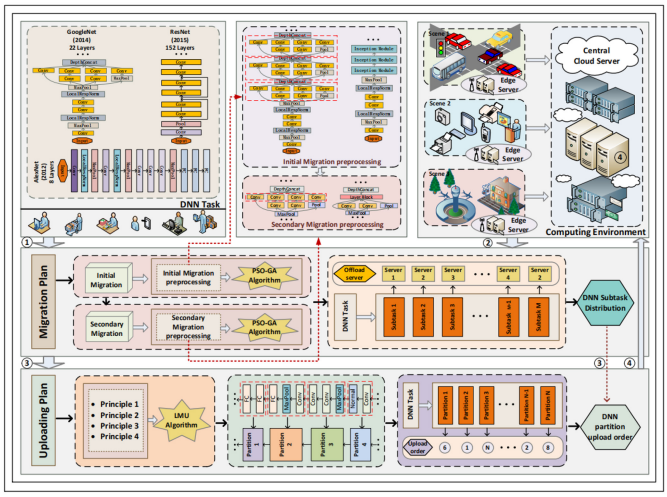


Fig. 1. The process of EosDNN offloading scheme. Among them, ① represents the parameters of the DNN model, and ② represents the environmental parameters of the local-edge-cloud collaborative environment. ③ means that after the migration plan determines the migration location of the DNN subtask, the uploading plan obtains the DNN partition and the upload order of DNN partition based on this. ④ represents the process of uploading the DNN partition to the computing environment.

generating efficient migration plans for the parallel processing of DNN models in resource-limited computing environments. Therefore, we propose a *PSO-GA algorithm* for DNN parallel migration in a multi-user local-edge-cloud collaborative environment with limited resources and then generate a migration plan. The *PSO-GA algorithm* uses individual local information and group global information to guide the search and has a fast convergence speed, which is suitable for efficiently generating a DNN migration plan. Finally, considering the diverse structure of the DNN model, in order to enhance the generalization ability of the migration plan, we proposed *two-step migration* to deal with the topological DNN model with inception modules.

B. Uploading Plan

Based on the *migration plan*, the migration position of each DNN layer is known. Then, we propose *LMU algorithm* to generate a DNN uploading plan, so as to obtain the DNN partition and determine the upload order of the DNN partition.

In Table IV, we notice that the DNN model is too large, so the transmission delay to upload the entire DNN model is too long. Before DNN model uploading is completed, DNN queries will only be run by the client, causing a lot of burden to the client. Due to the low computing power of the client, DNN inference efficiency is very low. Considering that the DNN partition granularity of the existing DNN uploading plan is too large, it is difficult to obtain better DNN query performance. In addition, the existing DNN uploading plans are applied from a single client to a single edge/cloud computing node, which does not conform to the trend of edge-cloud collaboration. Therefore, we proposed the *LMU algorithm* in the local-edge-cloud collaborative environment. As we all know, the DNN layer is the smallest granularity of the DNN model. Based on *LMU algorithm*, the DNN layer is merged with adjacent

layers to form a DNN partition, which avoids the excessive granularity caused by the whole DNN model partition in the existing work. In this case, the generation of fine-grained partitions can be maximized, and the performance of DNN queries can be optimized.

V. Proposed Migration Plan

We use the *PSO-GA algorithm* to analyze the subtask distribution in a multi-user local-edge-cloud collaborative environment, so as to obtain the first part of *EosDNN* offloading scheme, i.e., migration plan. The section includes three parts: Firstly, we preprocess the topological DNN model and then propose the *two-step migration* of the DNN model. Secondly, we briefly introduce the mathematical model of the migration plan. Finally, we consider the operation of *PSO-GA algorithm* under the migration plan.

A. Two-Step Migration

1) Two-Step Migration Preprocessing: We know that DNN models usually include chain DNN models and topological DNN models containing inception modules. Considering the data dependency between DNN subtasks and the migration efficiency of DNN models, it is usually necessary to preprocess the DNN model.

When the deviation between the out-degree of the predecessor and the in-degree of the successor under the inception module is 1, some scholars merge the two adjacent layers into a new layer. After preprocessing, the data dependency between the predecessor and the successor will disappear [30]. Some methods directly regard the inception module as a whole and convert the topological DNN model containing the inception module into a chain DNN model [35]. In addition, some scholars regard the inception module as a minimum cutting problem and use Boykov–Kolmogorov maximum flow algorithm, topological sorting algorithm, and other algorithms to perform a DNN partition [36]. However, the existing preprocessing topology DNN model is not suitable for generating fine-grained DNN partitions, and it is difficult to obtain a more efficient DNN migration plan and DNN uploading plan. Therefore, we design *two-step migration preprocessing* as follows.

- *Initial Migration Preprocessing:* For the topological DNN model, we first directly regard the inception module as a whole, and convert the topological DNN model into a chain DNN model, which ensures that the data transmission relationship of the inception module is not destroyed.
- *Secondary Migration Preprocessing:* Considering that the repeated transmission of the same input data will bring unnecessary transmission pressure to the wireless network, it will also bring higher transmission delay. We treat the DNN layers with the same input under the inception module as a whole and migrate them to the same edge/cloud server.

Taking Fig. 2 as an example, we regard *layer block A* as one layer during the *initial migration preprocessing*. In the *secondary migration preprocessing*, we regard *layer block B* and *layer block C* as one layer, respectively, and then split the inception module.

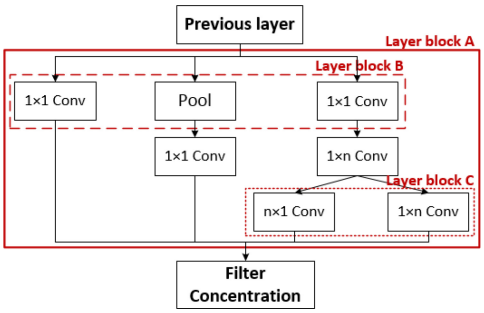


Fig. 2. Inception module.

2) Two-Step Migration: In order to process the topological DNN with inception module, based on the *two-step migration preprocessing*, we propose a *two-step migration* method as follows.

- *Initial Migration:* based on the *initial migration preprocessing*, we regard the inception module as one layer in the DNN model, thus transforming the topological DNN model into the chain DNN model, and then the chain DNN model is divided for the first time. In the *Initial Migration*, we can know the migration position of the input and output layers of the inception module.
- *Secondary Migration:* Based on the *secondary migration preprocessing*, we divide the inception module for the second time, migrate each branch of the inception module, and get the subtask distribution of the inception module under the cloud/edge server. At this time, the final migration delay of the inception module is the branch that takes the longest time, and the energy consumption and cost are the sums of all branches, respectively.

B. Migration Plan Analysis

In the local-edge-cloud collaborative environment, we can choose to perform subtasks D_{ij} locally or migrate to a cloud/edge server. We first calculate the transmission delay, execution delay, and waiting delay, respectively. Then we establish the migration delay formula of the DNN model under the *two-step migration*.

We define an indicative function $\mathbf{1}_d$, where d represents the migrating location of subtask D_{ij} in the local environment $\mathbb{M}$. For instance, for the indicative function $\mathbf{1}_{d\in[1,u]}$, if the actual migrating location of D_{ij} is r_d, when $d > u$, $\mathbf{1}_{d\in[1,u]} = 0$, otherwise, $\mathbf{1}_{d\in[1,u]} = 1$.

1) Execution Delay: After the input data required by subtask D_{ij} is transmitted to the cloud/edge server, the subtask starts to execute, and its execution delay T_{ij}^1 is as follows:

$$T_{ij}^1 = \frac{P_{ij}}{Q_{ij}^l}\mathbf{1}_{d\in[1,u]} + \frac{P_{ij}}{Q_{ij}^e}\mathbf{1}_{d\in[u+1,u+o]} + \frac{P_{ij}}{Q_{ij}^c}\mathbf{1}_{d\in[u+o+1,s]}, \tag{1}$$

where P_{ij} represents the number of CPU cycles to complete subtask D_{ij}. Q_{ij}^l, Q_{ij}^e and Q_{ij}^c indicate the computing power

of the client, edge server and cloud server where subtask D_{ij} is located, respectively.

2) Transmission Delay: Since the subtask D_{ij} under the task D_i is executed serially, we consider the data transmission between the server where subtask D_{ij-1} is located and the server where subtask D_{ij} is located. Let v denote the data transmission rate between servers, which can be defined as:

$$v = \begin{cases} \infty, & \text{if } r_d \text{ and } r_{d'} \text{ are connected } \& \ r_d = r_{d'}, \\ \eta, & \text{if } r_d \text{ and } r_{d'} \text{ are connected } \& \ r_d \neq r_{d'}, \\ 0, & \text{if} r_d \text{ and } r_{d'} \text{ are not connected.} \end{cases} \quad (2)$$

where r_d and $r_{d'}$ represent two servers in a local-edge-cloud collaborative environment, and η is a constant.

Therefore, the transmission delay T_{ij}^2 between subtasks D_{ij-1} and D_{ij} is calculated as follows:

$$T_{ij}^2 = \frac{g_{ij}}{v_{ij}} \mathbf{1}_{d\in[1,u]} + \frac{g_{ij}}{v_{ij}} \mathbf{1}_{d\in[u+1,u+o]} + \frac{g_{ij}}{v_{ij}} \mathbf{1}_{d\in[u+o+1,s]}, \quad (3)$$

where g_{ij} represents the transmission data between subtasks D_{ij-1} and D_{ij}, and v_{ij} represents the transmission rate between the server where subtask D_{ij-1} is located and the server where subtask D_{ij} is located.

3) Waiting Delay: The number of parallel pools represents the maximum number of subtasks that the cloud/edge server can execute concurrently, and each parallel pool can execute a DNN subtask. For example, if the number of parallel pools under a cloud/edge server is four, it means that the cloud/edge server can process up to four subtasks simultaneously. We assume that the number of parallel pools per server is limited, that is, computing resources are limited. Therefore, when formulating a migration plan, due to the limitation of the parallel pool, it is necessary to consider the waiting time due to subtask queuing under each cloud/edge server.

Let x_d^{pl}, x_d^{run} and x_d^{st} denote the number of parallel pools, the number of running subtasks, and the number of subtasks to be run under the server r_d, respectively. Obviously, $x_d^{run} \leq x_d^{pl}$. In addition, if $x_d^{st} \leq x_d^{pl}$, the queuing time $z_{ij} = 0$. If $x_d^{st} > x_d^{pl}$, we calculate the difference I_{ij} between the complete execution time of subtask being executed and the start execution time of subtask being queued. If $I_{ij} > 0$, it indicates that the subtask needs to queue, and the queuing time $z_{ij} = I_{ij}$. If $I_{ij} \leq 0$, indicating that the subtask does not need to wait, then the queuing time $z_{ij} = 0$. The waiting delay T_{ij}^3 can be expressed as:

$$T_{ij}^3 = z_{ij} \mathbf{1}_{d\in[1,u]} + z_{ij} \mathbf{1}_{d\in[u+1,u+o]} + z_{ij} \mathbf{1}_{d\in[u+o+1,s]}, \quad (4)$$

$$I_{ij} = T_{i'j'}^2 + T_{i'j'}^1 + z_{i'j'} - T_{ij}^2 + \left(t_{i'j'} - t_{ij}\right), \quad (5)$$

$$z_{ij} = \begin{cases} 0, & x_d^{st} \leq x_d^{pl}, \\ \max\{0, I_{ij}\}, & x_d^{st} > x_d^{pl}. \end{cases} \quad (6)$$

where $D_{i'j'}$ and D_{ij} respectively represent the subtask that ends first and the subtask that arrives first under the cloud/edge server. z_{ij} and $z_{i'j'}$ denote the queuing time of subtask D_{ij} and $D_{i'j'}$, respectively. $T_{i'j'}^1$ denotes the execution delay of subtask $D_{i'j'}$, T_{ij}^2 denotes the transmission delay between subtasks D_{ij-1} and D_{ij}, $T_{i'j'}^2$ denotes the transmission delay between subtasks $D_{i'j'-1}$ and $D_{i'j'}$, t_{ij} and $t_{i'j'}$ denote the time to start transmitting the input data of the subtask D_{ij} and $D_{i'j'}$, respectively.

C. Problem Formulation

For the topological DNN model containing the inception module, we consider the *two-step migration*.

- *Initial Migration:* The first step is to treat the inception module as a whole, and treat the inception module as a layer D_{ij} in the DNN model D_i. Our goal is to find the best migration plan with the lowest migration delay in the local-edge-cloud collaboration environment to meet the resource constraints of each server. Then, the optimization formula is as follows:

$$\begin{aligned} \min \ : \ & \sum_i \sum_j T_{ij}^1 + T_{ij}^2 + T_{ij}^3 \\ s.t. \ : \ & x_d^{run} \leq x_d^{pl} \end{aligned} \quad (7)$$

- *Secondary Migration:* After *initial migration*, the migration locations of the input and output layers of the inception module are available. Based on this, we migrate each branch of the inception module separately. It is important to note that the migration delay of the inception module is the single branch with the highest delay, as follows:

$$\begin{aligned} \min \ : \ & \sum_c \max_{j^*} \left\{ \sum_w T_{ij^*}^{cw1} + T_{ij^*}^{cw2} + T_{ij^*}^{cw3} \right\} \\ s.t. \ : \ & x_d^{run} \leq x_d^{pl} \end{aligned} \quad (8)$$

Some layers of tasks D_{ij} in the *initial migration* are inception modules, defined as $D_{ij^*}^{cw}$. For these layers, we perform a *secondary migration*, where *c* is the total number of inception modules, *w* is the number of layers in each branch of the inception module, j^* indicates that the *j*-th layer in the *initial migration* is the inception module. In addition, the formulas of $T_{ij^*}^{cw1}$, $T_{ij^*}^{cw2}$ and $T_{ij^*}^{cw3}$ are the same as the formulas of T_{ij}^1, T_{ij}^2 and T_{ij}^3.

D. PSO-GA Algorithm

The fundamental goal of the DNN migration plan is to find a mapping from $\mathbb{D}$ to $\mathbb{R}$ to minimize the delay of the migration plan. In addition, we need to note that finding the best mapping from $\mathbb{D}$ to $\mathbb{R}$ is an NP-hard problem [37]. Based on this, we use *PSO-GA algorithm* to get the best mapping from $\mathbb{D}$ to $\mathbb{R}$, that is, the migration plan. In order to explain more concisely, we mainly discuss Eq. (7) in *initial migration*. (Inception module migration in *secondary migration* is the same as the *initial migration* except for the difference of the fitness function).

1) Rationality: The *PSO algorithm* is a collaborative search algorithm that uses individual local information and group global information to guide the search at the same time. It has a faster convergence rate and is suitable for efficiently generating DNN migration plans. However, the local search ability of *PSO algorithm* is poor, and the search accuracy is not high enough to obtain an accurate optimal solution. The search performance of the *PSO algorithm* depends on the balance of its global exploration and local refinement, which depends to

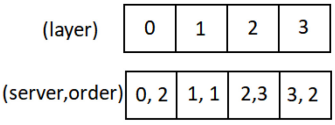
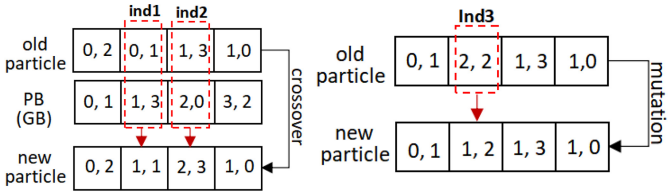


Fig. 3. An encoded particle for a DNN migration.



(a) Crossover operator for individual (social) cognition. (b) Mutation operator for inertia component.

Fig. 4. Update operator.

a large extent on the inertial component, individual cognition, and social cognition of the algorithm.

In order to avoid premature convergence of particles and improve the global search performance, the *PSO-GA algorithm* introduces the crossover operator and mutation operator of the *GA algorithm* when the particles are updated, so that the particles can gain the ability to explore new regions, thereby obtaining better global search capabilities. For individual cognitive components and social cognitive components, the crossover operator of *GA algorithm* is introduced to refresh the corresponding components. In addition, we combine the inertial component with the mutation operator of the *GA algorithm*, which can better balance the local search and the global search, and then obtain the optimal migration plan in local-edge-cloud collaborative computing environments.

2) Coding Strategy: A good coding strategy is a prerequisite for a reasonable migration decision, which usually needs to comply with the following three principles:

- *Completeness:* Each candidate solution can be coded as a particle.
- *Non-redundant:* Each candidate solution has only one corresponding coded particle.
- *Feasibility:* Each coded particle represents a candidate solution.

In Fig. 3, the *PSO-GA algorithm* uses a nested strategy to encode the migration problem of the DNN layer, where each particle represents a candidate solution for all DNNs, and the k-th particle H_k in the t-th iteration is described as:

$$H_k^t = \left(h_{k1}^t, h_{k2}^t, \ldots, h_{k\bar{\beta}}^t\right), \tag{9}$$

$$h_{kj}^t = \left(r_d, u_j\right)_{kj}^t, \tag{10}$$

where $\bar{\beta}$ is the total number of layers in all DNN models, h_{kj}^t means that in the t-th iteration, the server r_d will execute subtask D_{ij} in the order of u_j. $u_j = 0, 1, \ldots, \bar{\beta} - 1$ indicates the order of subtask D_{ij}.

3) Fitness Function: The fitness function is used to evaluate the performance of particles, and particles with lower fitness represent better candidate solutions. The purpose of this paper is to pursue the minimum migration delay, so we define the fitness function as follows:

$$f(H_k) = \sum_i \sum_j T_{ij}^1 + T_{ij}^2 + T_{ij}^3, \tag{11}$$

where $f(H_k)$ represents the migration delay in a multi-user local-edge-cloud collaborative environment. The lower the individual fitness, the less time it takes to complete all tasks, which means that it is less likely to be eliminated from the population.

4) Update Strategy: We know that *PSO algorithm* mainly includes inertia component, individual cognition, and social cognition, while the high quality of traditional *PSO algorithm* is easy to fall into the defect of local optimization. In order to avoid premature convergence and improve the global search performance, the *PSO-GA algorithm* introduces the crossover operator and mutation operator of the *GA algorithm* during particle updates. The crossover operator $p()$ (or $g()$) is shown in Fig. 4(a), randomly selecting the *ind1* and *ind2* positions in the old particle, and then replacing the segment between *ind1* and *ind2* with partial best *PB* (or global best *GB*) particles in the same interval. The mutation operator $M()$ is shown in Fig. 4(b), by randomly selecting the position in the particle and changing the corresponding server.

Iterative Update: When the i-th particle is in the t-th iteration, the iteration update is as follows:

$$H_k^t = \epsilon_2 \oplus g\left(\epsilon_1 \oplus p\left(w \oplus M\left(H_k^{t-1}\right), PB_k^{t-1}\right), GB^{t-1}\right). \tag{12}$$

Inertial Component: The mutation operator $M()$ is introduced to refresh the inertial component A_k^t.

$$A_k^t = w \oplus M\left(H_k^{t-1}\right) = \begin{cases} M\left(H_k^{t-1}\right), & r_1 < w, \\ H_k^{t-1}, & else. \end{cases} \tag{13}$$

Individual Cognition: Introduce crossover operator $p()$ to refresh individual cognition B_k^t.

$$B_k^t = \epsilon_1 \oplus p\left(A_k^t, PB^{t-1}\right) = \begin{cases} p\left(A_k^t, PB^{t-1}\right), & r_2 < \epsilon_1, \\ A_k^t, & else. \end{cases} \tag{14}$$

Social Cognition: Introduce crossover operator $g()$ to refresh social cognition C_k^t.

$$C_k^t = \epsilon_2 \oplus g\left(B_k^t, GB^{t-1}\right) = \begin{cases} g\left(B_k^t, GB^{t-1}\right), & r_3 < \epsilon_2, \\ B_k^t, & else. \end{cases} \tag{15}$$

where PB^{t-1} is the partial optimal solution in the $t-1^{th}$ iteration, and GB^{t-1} is the global optimal solution in the $t-1^{th}$ iteration. ϵ_1 and ϵ_2 represent acceleration coefficients. r_1, r_2 and r_3 represent random numbers between [0, 1]. In addition, the order of DNN layers u_j is not updated with iteration. ϵ_1^{st} and ϵ_1^{ed} are the start value and end value of ϵ_1, respectively. ϵ_2^{st} and ϵ_2^{ed} are the start value and end value of ϵ_2, respectively.

Algorithm 1: Migration Algorithm

Input: Local-edge-cloud collaborative environment parameters (bandwidth, server computing power, number of various servers), DNN model description (number of CPUs required for execution layer, data transfer volume).

Output: The optimal migration plan.

Preprocessing: obtain the current optimal allocation plan and the historical optimal plan through the GA algorithm, and update the *PSO algorithm* parameters.
for *particle* $k = 1$ **do**
  Initialize particle with crossover and mutation operations
  $T_i = \sum_j T_{ij}^1 + T_{ij}^2 + T_{ij}^2$
  Calculate the fitness of particle k and
  set $PB_k = H_k$
  $GB = \min\ PB_k$
  for $k = 1$ *to* N **do**
    Update particle with crossover and mutation operations
    Calculate the fitness of particle
    $fitness(H_k) = \sum_i \sum_j T_{ij}^1 + T_{ij}^2 + T_{ij}^3$
    if $fitness(H_k) < fitness(PB_k)$ **then**
      $PB_k = H_k$
    if $fitness(PB_k) < fitness(GB)$ **then**
      $GB = PB_k$
final
return *GB* as migration plan

5) Mapping of Particles and DNN Migration: The k^{th} particle of the *PSO-GA algorithm* is in the local-edge-cloud collaborative environment, the partial optimal solution PB_k is the relative optimal distribution of subtasks, and the global optimal solution *GB* indicates the optimal distribution of subtasks, particle H_k represents the distribution and order information of subtasks. We set up *N* particles. The pseudocode of the algorithm is as shown in Algorithm 1. A more preferred subtask allocation scheme is generated by the *GA algorithm* as the initial solution, and then the *PSO algorithm* is applied to find a subtask migration plan close to the optimal solution in the solution space, then we combine crossover and mutation operations to update.

6) Parameter Settings: We know that the inertia weight *w* affects the convergence and search ability of *PSO algorithm*. The *PSO-GA algorithm* designs an adjustment mechanism that adapts to the nonlinear migration characteristics, and considers an adjustment mechanism that can adaptively adjust the search capability according to the current particle:

$$w = w_{max} - (w_{max} - w_{min}) \times e^{\frac{d\left(H_k^{t-1}\right)}{d\left(H_k^{t-1}-1.01\right)}}, \quad (16)$$

where w_{max} and w_{min} are the maximum and minimum values of *w* in the initialization phase, respectively. $d(H_k^{t-1}) = \frac{div(GB^{t-1}, H^{t-1})}{|C|}$, where $div(GB^{t-1}, H^{t-1})$ denotes the coordinate difference between the global best particle GB^{t-1} and the current particle H^{t-1} in the $t-1^{th}$ iteration, i.e., the lower $div(GB^{t-1}, H^{t-1})$ is, the closer GB^{t-1} and H^{t-1} are. In addition, the search capability is adaptively adjusted according to the difference between the current particle and the global best particle.

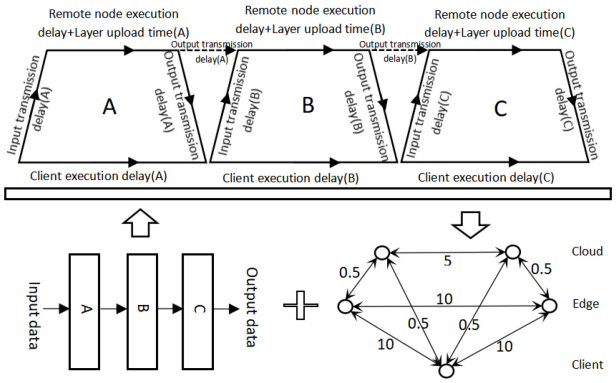


Fig. 5. Schematic diagram of DNN layer uploading.

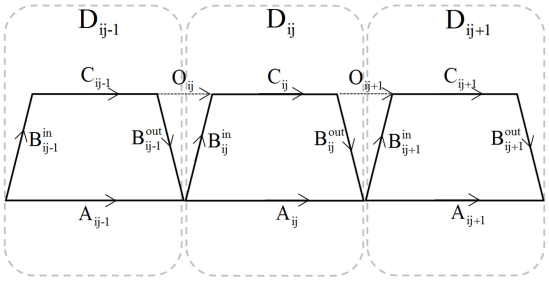


Fig. 6. Diagram of data transmission. B_{ij}^{in}, B_{ij}^{out} represent input and output data transmission delays of layers D_{ij} between the client and server, respectively. O_{ij} represents the input data transmission of layer D_{ij} between two servers. C_{ij} and A_{ij} represent the execution delay of layer D_{ij} on the client and server, respectively.

VI. Proposed Uploading Plan

Through the migration plan, we can know the migration position of each DNN layer. Based on this, we propose a *layer merge uploading algorithm* to obtain DNN partitions and formulate the upload order of DNN partitions, thereby obtaining the second part of the *EosDNN* offloading scheme, i.e., the uploading plan. In Fig. 5, we briefly introduce the data flow of the DNN model.

In this section, we first give some common definitions. Then, we design four principles for DNN model uploading. Finally, we propose an *LMU algorithm* to upload the DNN model according to four principles.

A. Common Definition

We give several common definitions: *Definition 1 (merged layer): Merged layer* refers to the combination of several adjacent layers. According to Fig. 6, when the layer D_{ij} and the layer D_{ij+1} are uploaded to the cloud/edge server as a whole, it is called a *merged layer*.

Definition 2 (DNN partition): DNN partition represents the division of the DNN model, including the DNN layers and merged layers.

Definition 3 (Uploading Method Definition): We divide uploading methods into *Independent Uploading* and *Non-independent Uploading*, in which *Non-independent Uploading* can be divided into three cases. The detailed definition is as follows.

- *Independent Uploading:* When a DNN partition is uploaded to the cloud/edge server, but the input DNN partition and output DNN partition are not uploaded.
- *A-Non-Independent Uploading:* For a DNN partition, if the input DNN partition has been uploaded to the cloud/edge server, but the output DNN partition has not been uploaded.
- *B-Non-Independent Uploading:* For a DNN partition, if the output DNN partition has been uploaded to the cloud/edge server, but the input DNN partition has not been uploaded.
- *C-Non-Independent Uploading:* For a DNN partition, both the input DNN partition and output DNN partition have been uploaded to the cloud/edge server.

In Fig. 6, based on [32], we give definitions for four different conditions related to delay.

Definition 4 [Total Delay Advantage (TDA)]: The delay advantage obtained by executing all DNN partitions on a cloud/edge server compared to executing all DNN partitions locally.

- *Example 1:* For D_{ij-1}, D_{ij} and D_{ij+1} upload in turn, the corresponding *TDA*s are TDA'_{ij-1}, TDA'_{ij} and TDA'_{ij+1}, respectively, where $TDA'_{ij-1} = A_{ij-1} - C_{ij-1} - B^{in}_{ij-1} - B^{out}_{ij-1}$, $TDA'_{ij} = \sum_{a=j-1}^{j}(A_{ia} - C_{ia}) - O_{ij} - B^{in}_{ij-1} - B^{out}_{ij}$ and $TDA'_{ij+1} = \sum_{b=j-1}^{j+1}(A_{ib} - C_{ib}) - \sum_{c=j}^{j+1} O_{ic} - B^{in}_{ij-1} - B^{out}_{ij+1}$.
- *Example 2:* For D_{ij-1}, D_{ij+1} and D_{ij} upload in turn, the corresponding *TDA*s are TDA''_{ij-1}, TDA''_{ij+1} and TDA''_{ij}, respectively, where $TDA'_{ij-1} = A_{ij-1} - C_{ij-1} - B^{in}_{ij-1} - B^{out}_{ij-1}$, $TDA'_{ij+1} = A_{ij+1} - C_{ij+1} - B^{in}_{ij+1} - B^{out}_{ij+1}$ and $TDA'_{ij} = \sum_{b=j-1}^{j+1}(A_{ib} - C_{ib}) - \sum_{c=j}^{j+1} O_{ic} - B^{in}_{ij-1} - B^{out}_{ij+1}$.

Definition 5 [Layer Delay Advantage (LDA)]: LDA refers to the optimization of the *TDA* after uploading each DNN partition. We define the *TDA* change caused by each DNN partition upload as *LDA*. In addition, we record the *LDA* under the *Independent Uploading* mode as LDA^*.

- *Example 3:* For D_{ij-1}, D_{ij} and D_{ij+1} upload in turn, the corresponding *LDA*s are LDA'^*_{ij-1}, LDA'_{ij} and LDA'_{ij+1}, where $LDA'^*_{ij-1} = TDA'_{ij-1}$, $LDA'_{ij} = TDA'_{ij} - TDA'_{ij-1}$ and $LDA'_{ij+1} = TDA'_{ij+1} - TDA'_{ij}$.
- *Example 4:* For D_{ij-1}, D_{ij+1} and D_{ij} upload in turn, the corresponding *LDA*s are LDA''^*_{ij-1}, LDA''_{ij} and LDA''^*_{ij+1}, where $LDA''^*_{ij-1} = TDA''_{ij-1}$, $LDA''^*_{ij+1} = TDA''_{ij+1}$ and $LDA''_{ij} = TDA''_{ij} - TDA''_{ij-1} - TDA''_{ij+1}$.
- *Example 5:* D_{ij-1} and D_{ij} are merged to upload, then the LDA^* of the merged layer is $A_{ij-1} + A_{ij} - (C_{ij-1} + O_{ij} + C_{ij}) - B^{in}_{ij-1} - B^{out}_{ij}$.

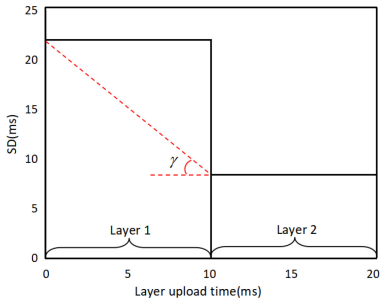


Fig. 7. Geometric representation of *ELDA*.

Definition 6 [System Delay (SD)]: As DNN partitions are uploaded, the delay for completing DNN model queries will change constantly. We define the delay of completing the DNN model query as *SD*. In Fig. 8, we denote the sum of *SD* changes during the continuous upload process of the DNN partition as *SSD*.

- *Example 6:* In Fig. 6, if the layer D_{ij-1} and D_{ij+1} are executed on the cloud/edge server, and layer D_{ij} is executed locally. In addition, first upload layer D_{ij-1} and then upload layer D_{ij+1}. The layer upload delay is t_{ij-1} and t_{ij+1} respectively.
  The *SD* before uploading layer D_{ij-1} is defined as $\overline{SD} = A_{ij-1} + A_{ij} + A_{ij+1}$.
  The *SD* after uploading layer D_{ij-1} is $SD_{ij-1} = B^{in}_{ij-1} + C_{ij-1} + B^{out}_{ij-1} + A_{ij} + A_{ij+1}$.
  The *SD* after uploading layer D_{ij+1} is $SD_{ij+1} = B^{in}_{ij-1} + C_{ij-1} + B^{out}_{ij-1} + A_{ij} + B^{in}_{ij+1} + C_{ij+1} + B^{out}_{ij+1}$.
  The *SSD* during uploading layers D_{ij-1} and D_{ij+1} is defined as $SSD = \overline{SD} \times t_{ij-1} + SD_{ij-1} \times t_{ij+1}$.

Definition 7 [Migration Delay (MD)]: It refers to the delay obtained by *PSO-GA algorithm*, which is also equal to *SD* after uploading the final DNN partition.

Definition 8 [Efficiency of Layer Delay Advantage (ELDA)]: The ELDA of DNN partition under unit initial input data is defined as:

$$ELDA = \frac{LDA/DNN\ partition\ upload\ delay}{initial\ input\ data\ size}. \quad (17)$$

In Fig. 7, *ELDA* can be regarded as $\tan\gamma$, meaning that *SD* changes during the DNN partition upload.

B. Uploading Principle Analysis

In this section, we put forward four uploading principles of the uploading plan. Firstly, we briefly introduce the four uploading principles. Secondly, we analyze the operation process of the four principles. Finally, the formulation principle of these four principles is demonstrated in detail.

1) Uploading Principle: The contents of the four uploading principles are as follows.

- *Principle 1:* DNN partitions are uploaded in descending order of *ELDA*.
- *Principle 2:* After pre-uploading the *size* $= 0$ layer under the merged layer with $LDA^* > 0$, the *size* $= 0$ layer will be executed directly. However, after pre-uploading the *size* $= 0$ layer under the merged layer with $LDA^* < 0$,

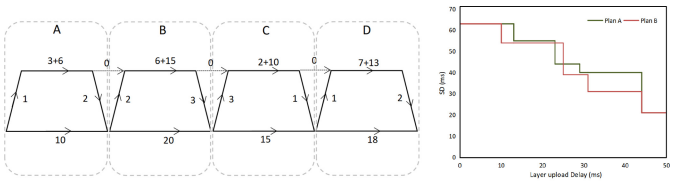
it will not be executed until the rest of the merged layer is uploaded.

- *Principle 3:* Based on *Principle 1*, when several adjacent layers with $LDA^* > 0$, upload these layers directly (*Direct Uploading*); when part of the DNN partition with $LDA^* < 0$, choose to combine with adjacent layers to form a new merged layer with $LDA^* > 0$, and then upload the new merged layer(*Merge uploading*).
  For *Merge uploading*, we proposed *Merge Rules* to handle the layer with $LDA^* < 0$. In other words, if there is a layer with $LDA^* < 0$, merge it with the adjacent layer and form a merged layer with $LDA^* > 0$ according to the *Merge Rules*. The details are as follows:
  - For two merged layers with $LDA^* > 0$ composed of input or output adjacent layers, we choose the merged layer with a higher *ELDA*.
  - The merged layer can be composed of the input or output adjacent layer. For the LDA^* of the two merged layers, if one is greater than 0 and the other is less than 0, then the merged layer with $LDA^* > 0$ is selected.
  - When two merged layers with $LDA^* < 0$ are constituted by the input or output adjacent layer, then we choose to form a merged layer with output adjacent layers. If the merged layer with $LDA^* < 0$, then continue to find the adjacent layers of the merged layer based on the above merging rule until a new merged layer with $LDA^* > 0$ is formed.
- *Principle 4:* According to *principle 1*, the merged layer was further divided and uploaded. In other words, when uploading a merged layer, follow *principle 1* to upload the *sub-partitions* (including the layers and smaller merged layers under the merged layer) with $LDA^* > 0$ separately, and then upload the rest of the merged layer at once.

2) Operation of the Principle 1-4: We briefly introduce the operation of *Principle 1-4*, from which we can easily discover the relationship between the four principles: *Principle 2* is the premise; *Principle 1* is the basis of *Principle 3* and *Principle 4*.

- First of all, we will upload the *size* $= 0$ layer in advance based on *Principle 2*.
- Secondly, we propose a *Merge Rules* based on *Principle 3* to combine the $LDA^* < 0$ layer with its adjacent layers to obtain the merged layer with $LDA^* > 0$ and then get the DNN partition under the uploading plan.
- Thirdly, we will formulate uploading plans of the DNN partition based on *Principle 1*.
- Finally, we combine *Principle 4*, and formulate uploading plans of the *sub-partition* with $LDA^* > 0$ based on *Principle 1*.
  It should be noted that when formulating an uploading plan of the *sub-partition* with $LDA^* > 0$, the upload order of the layers and merged layers in the entire DNN model remains unchanged, but the merged layer is uploaded in a more fine-grained partition.

3) Principle Theory: In this part, we will explain the *Principle 1-4* in detail.



(a) Schematic diagram of data transmission (b) *SD*

Fig. 8. The performance of *SD* under different upload orders.

TABLE II
THE *SD* OF PLAN A AND PLAN B

Order	D →	C→	A →	B →	End
LUD (Plan A)	$0 \sim 13$	$13 \sim 23$	$23 \sim 29$	$29 \sim 44$	$44 \sim 50$
SD (Plan A)	63	55	44	40	21
Order	C →	B→	A →	D →	End
LUD (Plan B)	$0 \sim 10$	$10 \sim 25$	$25 \sim 31$	$31 \sim 44$	$44 \sim 50$
SD (Plan B)	63	54	39	31	21

TABLE III
DETAILED STEPS FOR PLAN B

Step	A	B	C	D	UL
1	0.67	0.60	0.90	0.62	C
2	0.67	1.00	×	0.77	B
3	1.33	×	×	0.77	A
4	×	×	×	0.77	D

Remark (Principle 1): Fig. 8(b) shows the *SD* changes for different upload orders under Fig. 8(a). Table II describes the layer upload delay (LUD) and system delay (*SD*) of uploading the DNN layer in different orders. Table III describes how to determine the upload order according to *Principle 1*.

- *Plan A:* It represents the *SD* changes when uploading the layer in the order of D→ *C* → *A* →B under the random upload layer mode.
- *Plan B:* It represents the *SD* changes when uploading the layer in the order of B→ *D* → *C* →A under *Principle 1*.

In Fig. 8, if the area of the plan (i.e., *SSD*) is smaller, it means that the *SD* is lower in the continuous process of layer upload, which means that the uploading plan is better. Since *SSD*(*Plan B*) $= 2203$, *SSD*(*Plan A*) $= 2359$, Plan B is obviously better than Plan A, which can continuously bring lower *SD* changes with continuous uploading of layers.

Remark (Principle 2): We discuss the corresponding *LDA* according to the upload order of DNN partitions.

- *Under the Same Server:* In Fig. 6, for adjacent layers under the same server, the data transmission delay $O_{ij} = O_{ij+1} = 0, B_{ij-1}^{out} = B_{ij}^{in}, B_{ij}^{out} = B_{ij+1}^{in}$.
  Independent Uploading: When layer D_{ij} is uploaded to the cloud/edge server, but D_{ij-1} and D_{ij+1} are not uploaded. The *LDA* of layer D_{ij} is as follows:

$$LDA_{ij}^{*} = A_{ij} - C_{ij} - B_{ij}^{in} - B_{ij}^{out}. \tag{18}$$

  A-Non-Independent Uploading: If layer D_{ij-1} has been uploaded to the cloud/edge server, but layer D_{ij+1} has not been uploaded. The *LDA* of layer D_{ij} can be defined as follows:

$$LDA_{ij}^{1} = A_{ij} - C_{ij} + B_{ij}^{in} - B_{ij}^{out}. \tag{19}$$

B-Non-Independent Uploading: Layer D_{ij+1} has been uploaded to the cloud/edge server, but layer D_{ij-1} has not been uploaded. The *LDA* of layer D_{ij} can be shown as:

$$LDA_{ij}^{2} = A_{ij} - C_{ij} - B_{ij}^{in} + B_{ij}^{out}, \tag{20}$$

C-Non-Independent Uploading: Both layers D_{ij-1} and D_{ij+1} have been uploaded to the cloud/edge server. The *LDA* of layer D_{ij} can be defined as:

$$LDA_{ij}^{3} = A_{ij} - C_{ij} + B_{ij}^{in} + B_{ij}^{out}. \tag{21}$$

- *Under Different Servers:* In Fig. 6, for adjacent layers under different servers, the data transmission delay $O_{ij}, O_{ij+1} \neq 0$, where $O_{ij} \leq B_{ij-1}^{out}$, $O_{ij+1} \leq B_{ij+1}^{in}$ according to the bandwidth.
Independent Uploading: When layer D_{ij} is uploaded to the cloud/edge server, but D_{ij-1} and D_{ij+1} are not uploaded, we define *LDA* of this layer as:

$$\overline{LDA_{ij}^{*}} = A_{ij} - C_{ij} - B_{ij}^{in} - B_{ij}^{out}. \tag{22}$$

A-Non-Independent Uploading: If layer D_{ij-1} has been uploaded to the cloud/edge server, but layer D_{ij+1} has not been uploaded, we define *LDA* of layer D_{ij} as:

$$\overline{LDA_{ij}^{1}} = A_{ij} - C_{ij} + B_{ij-1}^{out} - B_{ij}^{out} - O_{ij}. \tag{23}$$

B-Non-Independent Uploading: If layer D_{ij+1} has been uploaded to the cloud/edge server, but layer D_{ij-1} has not been uploaded, *LDA* of layer D_{ij} can be expressed as:

$$\overline{LDA_{ij}^{2}} = A_{ij} - C_{ij} - B_{ij}^{in} + B_{ij+1}^{in} - O_{ij+1}. \tag{24}$$

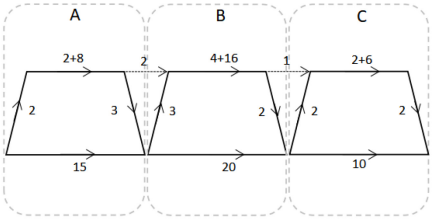
C-Non-Independent Uploading: When both layers D_{ij-1} and D_{ij+1} have been uploaded to the cloud/edge server, *LDA* of layer D_{ij} can be calculated by:

$$\overline{LDA_{ij}^{3}} = A_{ij} - C_{ij} + B_{ij+1}^{in} + B_{ij-1}^{out} - O_{ij} - O_{ij+1}. \tag{25}$$

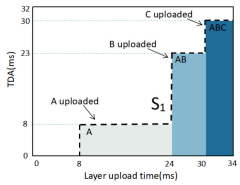
According to Eqs. (18)–(21) and Eqs. (22)–(25), we compare the relationship between the independent upload *LDA* and the non-independent upload *LDA* under the same server and different servers, respectively, and we can find $LDA_{ij}^{*} \leq LDA_{ij}^{u}, \overline{LDA_{ij}^{*}} \leq \overline{LDA_{ij}^{u'}}$, $u, u' = \{1, 2, 3\}$. From this, we can prove the rationality of *Principle 1*. At the same time, we will get a higher merged layer uploading advantage efficiency (*ELDA*) without increasing the layer upload delay.

Remark (Principle 3): We compare the *TDA* changes of several adjacent layers with $LDA^{*} > 0$ (*Type-1*) and part of the DNN partition with $LDA^{*} < 0$ (*Type-2*) under *Direct Uploading* and *Merge uploading*. The area S_n, $n = \{1, 2, 3, 4\}$ shows the sum of *TDA* changes during the continuous upload of the DNN layer in different upload methods. The details are as follows:

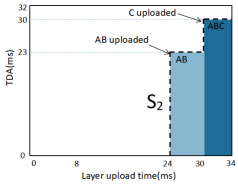
- *Direct Uploading (Type-1):* In Fig. 9(b), the area $S_1 = 386$, if layers A and B are uploaded directly in the order of A→ *B* →C.
- *Merge uploading (Type-1):* In Fig. 9(c), the area $S_2 = 258$, if layers A and B are merged and uploaded in the order of (AB)→C.



(a) Schematic diagram of data transmission

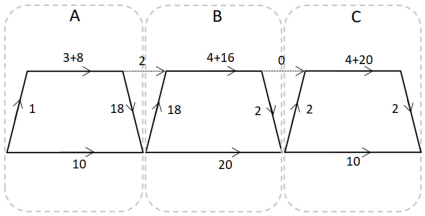


(b) Direct Uploading.

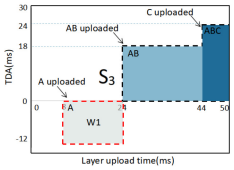


(c) Merge uploading.

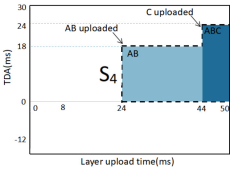
Fig. 9. The *TDA* performance of the layer with $LDA^{*} > 0$.



(a) Schematic diagram of data transmission



(b) Direct Uploading.



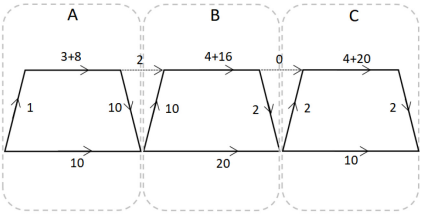
(c) Merge uploading.

Fig. 10. The *TDA* performance of the layer with $LDA^{*} < 0$.

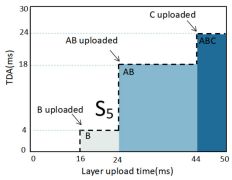
- *Direct Uploading (Type-2):* In Fig. 10(b), the area $S_3 = 312$, if layers A and B are uploaded directly in the order of A→ *B* →C. (The value of W_{max} is negative.)
- *Merge uploading (Type-2):* In Fig. 10(c), the area $S_4 = 504$, if layers A and B are merged and uploaded in the order of (AB)→C.

In Fig. 9, $S_1 > S_2$ shows that for several adjacent layers with $LDA^{*} > 0$, *Merge uploading* is worse than *Direct Uploading*. In Fig. 10, $S_3 < S_4$, which means that if the layer with $LDA^{*} < 0$, *TDA* of this layer will be ignored during the uploading process of the merged layer, so *Merge uploading* is a better uploading method.

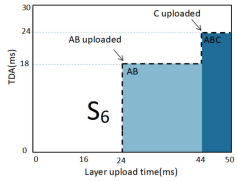
Remark (Principle 4): In Fig. 11(a), AB is a merged layer. We discuss the *TDA* changes brought about by the different upload methods of the merged layer. Fig. 11(b) shows that the merged layer is uploaded according to *Principle 1*. The layer B with $LDA^{*} > 0$ is uploaded first before the overall merged layer is uploaded, and $S_5 = 536$ indicates that *TDA* changes during the layer upload process. Fig. 11(c) shows that the entire merged layer AB is uploaded directly, and $S_5 = 504$ indicates that *TDA* changes during the layer upload process.



(a) Schematic diagram of data transmission



(b) Upload the merged layer based on *Principle 1*.



(c) Upload the merged layer directly.

Fig. 11. The *TDA* performance of the merged layer under different upload methods.

In Fig. 11, $S_5 > S_6$, indicating that uploading the merged layer according to *Principle 4* is better than uploading the entire merged layer directly.

C. Layer Merge Uploading Algorithm

Based on the migration plan, we apply *Principle 1-4* to create an *LMU algorithm* and then generate an uploading plan. Algorithm 2 shows the process of the algorithm, which includes five steps.

- *Step 1:* By using *PSO-GA algorithm* to generate a migration plan, thereby obtaining the migration position of DNN layer on the server (***PSO-GA algorithm*: line 1**).
- *Step 2:* In order to achieve non-independent uploading as much as possible, we consider uploading the *size* $= 0$ layer according to *Principle 2* (***Principle 2*: line 2**).
- *Step 3:* We consider processing layers with different LDA^* types: if $LDA^* > 0$, then it will not actively form a new merged layer with the adjacent DNN partition; if $LDA^* < 0$, it will actively form a new merged layer with the adjacent DNN partition according to *Merge Rules*. Thus, we can obtain DNN partitions. (***Principle 3*: lines 6∼11**).
- *Step 4:* Based on *Principle 1*, we obtain the upload order of DNN partition (***Principle 1*: line 4**).
- *Step 5:* According to *principle 1*, we further formulated the upload order of the sub-partitions with $LDA^* > 0$ under the merge layer (***Principle 4*: line 12**).

In addition, it is important to note that during the uploading process, we upload the inception module as one layer for DNN overall upload, rather than uploading the inception module step by step.

VII. Performance Evaluation

In this section, we evaluate the *EosDNN* offloading scheme in terms of the migration delay and DNN query performance.

A. Experimental Setup

We build a local-edge-cloud collaborative environment $\mathbb{R} = \{r_1, r_2, \ldots, r_{14}\}$, where the first four belong to the clients,

Algorithm 2: Layer Merge Uploading Algorithm

Input: Local-edge-cloud collaborative environment parameters (Bandwidth, server computing power, number of various servers), DNN model description (number of CPUs required for execution layer, data transfer, layer size).

Output: Optimal DNN uploading plan

1 .
2 Use *PSO-GA algorithm* to find the optimal migration plan;
3 Priority uploading *size* $= 0$ layer according to *Principle 2*;
4 **for** *task* D_i $i = 1, 2, 3, \ldots, \alpha$ **do**
5 Upload DNN partition under DNN model based on *Principle 1*.
6 **for** *layer* D_{ij} $j = 1, 2, 3, \ldots, \beta_i$ **do**
7 **if** *layer* $LDA^* < 0$ **then**
8 Form a merged layer with input or output adjacent layers, respectively.
9 **while** *merged layer with* $LDA^* > 0$ **do**
10 Select the adjacent layer with the highest *ELDA* to form a merged layer.
11 **while** *merged layer* $LDA^* < 0$ **do**
12 Form a merged layer with $j + 1^{th}$ layers.
13 Upload *sub-partitions* with $LDA^* > 0$ based on *Principle 1*.
14 **final**
15 **return** uploading plan

TABLE IV
DNN Model Parameters

Model	Params (M)	GFLOPs	Model size (MB)
Alexnet	62.1	0.7	237
GoogleNet	7.1	1.6	27
VGG	143.7	19.6	548
Resnet	44.6	7.6	170

the last five belong to the cloud servers, and the remaining five belong to the edge servers. The number of parallel pools of clients, edge servers, and cloud servers in the local-edge-cloud collaborative environment is 1, 2, and 8, respectively. We set the bandwidths between the client and the edge server to 10MB/s, the client and the cloud server to 0.5 MB/s, the edge server and the cloud server to 0.5 MB/s, the cloud server and the cloud server to 5 MB/s, the edge server and the edge server to 10 MB/s. The CPU processing capacity of the client, edge server, and cloud server is set as 1.1∼2.3 GHz, 4.2∼18.3 GHz, and 40∼120 GHz, respectively. The above-mentioned parameter setting is based on literature [30], [38], [39].

Following [30], [40], we set $\epsilon_1^{st} = 0.9$, $\epsilon_1^{ed} = 0.2$, $\epsilon_2^{st} = 0.4$, $\epsilon_2^{ed} = 0.9$, $w_{max} = 0.8$, $w_{min} = 0.2$, $N = 100$ and the number of iteration to 600. We adopt four different types of DNN models, namely, AlexNet [41], VGG [42], GoogleNet [43] and ResNet [44], where the model size is described in

TABLE V
LAYER SIZE OF THE ALEXNET MODEL

Layer	size (MB)	Layer	size (MB)	Layer	size (MB)	Layer	size (MB)
1	0.133300781	5	0	9	3.375976563	13	15.6288147
2	0	6	0	10	0		
3	0	7	3.376464844	11	144.015625		
4	2.344726563	8	5.063964844	12	64.015625		

TABLE VI
LAYER SIZE OF THE VGG MODEL

Layer	size (MB)	Layer	size (MB)	Layer	size (MB)	Layer	size (MB)
1	0.006835938	7	1.125976563	13	9.001953125	19	9.001953125
2	0.140869141	8	2.250976563	14	9.001953125	20	9.001953125
3	0	9	2.250976563	15	9.001953125	21	0
4	0.281738281	10	2.250976563	16	0	22	392.015625
5	0.562988281	11	0	17	9.001953125	23	64.015625
6	0	12	4.501953125	18	9.001953125	24	15.6288147

Table IV, the layer sizes for AlexNet and VGG are shown in Table V and Table VI, respectively. Other DNN model information, such as the transmission data between subtasks, and the framework of the DNN model, can be viewed in the file.[1]

We distribute tasks under each client, after the client connects to the cloud/edge server, the client executes the *PSO-GA algorithm* to create a migration plan and then executes the *LMU algorithm* to create an uploading plan. To reveal the effectiveness of the proposed *EosDNN* offloading scheme, we compare various migration plans and uploading plans in local-edge-cloud collaborative environments, which are listed as follows.

- *Non-Uploading Algorithm (Uploading Plan) [10]:* Once the distribution of the DNN layer is determined, the DNN layer uploaded to the cloud/edge server will be directly uploaded as a whole.
- *IONN Algorithm (Uploading Plan) [3]:* It applies the shortest path method and penalty factor method to upload DNN partitions from a single client to a single edge server, thereby dividing the DNN model and creating an uploading plan.
- *Recursive-Efficiency Algorithm (Uploading Plan) [32]:* It uses the shortest path method and efficiency-based recursive partitioning method to upload DNN partitions from a single client to a single edge server, thereby dividing the DNN model and create an uploading plan.
- *LMU Algorithm (Uploading Plan Under EosDNN):* We use *PSO-GA algorithm* to obtain the DNN layer distribution, and combine the *Principle 1-4* to propose a *LMU algorithm*, which can be applied to generate a more fine-grained DNN uploading plan in a multi-server environment.
- *Greedy Algorithm (Migration Plan) [45]:* This is a common method to find the optimal migration plan, which generally applies the greedy principle to select the best choice in the current state, and hopes to stack the final results together.
- *GA Algorithm (Migration Plan) [23]:* It imitates the natural evolution process, adopts the principle of crossover and mutation, and then carries out genetic iterations to produce a new uploading plan.
- *PSO-GA Algorithm (Migration Plan Under EosDNN) [30]:* The *PSO-GA algorithm* combines the crossover operator and the mutation operator under *GA algorithm*, which can better balance the local search and the global search, so as to obtain the global optimal DNN migration plan. In the *EosDNN* offloading scheme, we adopt a *two-step migration preprocessing* method to preprocess the DNN model.
- *prePSO Algorithm (Migration Plan) [30]:* Combine two adjacent layers into a new layer when the deviation between the out-degree of the predecessor and the in-degree of the successor is 1. After the above DNN model preprocessing is completed, *PSO-GA algorithm* is performed.

B. Partitioning Behavior

As we all know, smaller partitions can be uploaded to the cloud/edge server faster, which helps to obtain a greater additional layer delay advantage.

It is found from Fig. 12 that the proposed *EosDNN* offloading scheme can divide the largest partition into smaller partitions. Compared with *IONN algorithm* and *Recursive-efficiency*, *LMU algorithm* can significantly reduce the maximum distribution ratio for different types of DNN. We know that the DNN layer is the finest-grained DNN partition. Compared with the existing uploading plan, *LMU algorithm* can generate a finer-grained DNN partition uploading plan by combining the DNN layer and adjacent layers to form DNN partitions, avoiding the problem of excessive granularity of DNN partitions caused by direct segmentation of the whole DNN model.

[1] https://github.com/LinBin403/dataset-for-our-research

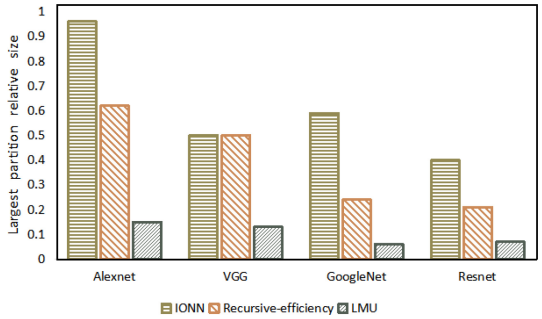
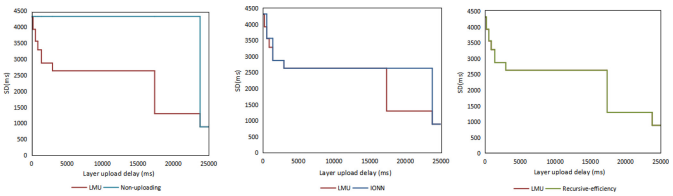
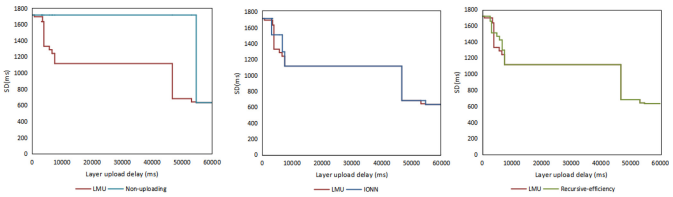


Fig. 12. The relative size of the largest partition compared to that of the entire model of various DNN models under different uploading plans.



(a) Non-uploading algorithm (b) IONN algorithm (c) Recursive-efficiency algorithm

Fig. 13. Comparison of total *SD* between EosDNN and other uploading plans with AlexNet.



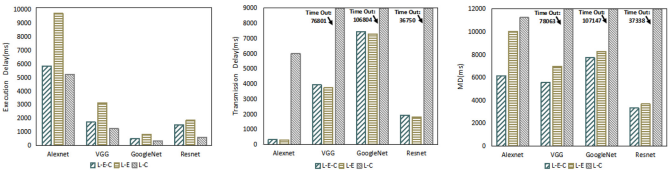
(a) Non-uploading algorithm (b) IONN algorithm (c) Recursive-efficiency algorithm

Fig. 14. Comparison of total *SD* between EosDNN and other uploading plans with VGG.

C. DNN Query Performance

We tracked the *SD* changes of repeated queries of Alexnet and VGG. As shown in Figs. 13-14, based on the migration plan, we can know the distribution of layers under servers. Based on this, we compared the *SD* changes of *non-uploading algorithm*, *IONN algorithm*, *efficiency-based uploading*, and *LMU algorithm*, respectively, under Alexnet and VGG for repeated queries. Among them, the abscissa represents the upload delay of the DNN model, and the ordinate represents the *SD* changes with the DNN partitions uploaded. We observe that the LDA^* of Alexnet is mostly positive, while the LDA^* of VGG is mostly negative. Both cover all types of DNN uploading. In addition, the layer upload delay is relatively long, and the model structure is relatively simple, making it a more suitable observation object. In particular, we notice that when the DNN model is large, the layer upload delay is much higher than the *SD* of the DNN model.

In Fig. 13(a), compared with the *non-uploading algorithm*, the *SSD* of *layer merge uploading algorithm* has been optimized by 45.16%. It can be observed from Fig. 13(b) that in the interval of 234.47∼572.12 ms and 17380.56∼23782.12 ms, the *SSD* of *LMU algorithm* is 13.41% lower than that of *IONN algorithm* schemes. In Fig. 13(c), since the $size = 0$ layer under merged layer with $LDA^* > 0$ is pre-uploaded, the cloud/edge server is still not in the "idle period" before uploading the first $size \neq 0$ layer. Therefore, the DNN query is executed by the local and cloud/edge servers in coordination to gain the layer delay advantage brought by the $size = 0$ layer, so *LMU algorithm* obtains a lower delay than *efficiency-based uploading*. Although the *SSD*s of these two schemes are similar under Alexnet, which is caused by many layers with $LDA^* > 0$ under the Alexnet model.



(a) Execution Delay (b) Transmission Delay (c) Total Delay

Fig. 15. The performance of various DNNs under different computing environments under Alexnet, VGG, Googlenet and Resnet, each with 12 tasks.

In Fig. 14(a), it is obvious that the fine-grained partitioned uploading plan adopted by the *LMU algorithm* can obtain a lower *SSD*. In Fig. 14(b), the *SSD* of *LMU algorithm* is lower than that of *IONN algorithm* during continuous uploading of DNN partitions. This advantage is mainly caused by the more fine-grained DNN partition and pre-uploading of the $size = 0$ layer under the merged layer with $LDA^* > 0$. In Fig. 14(c), the *SSD* of *LMU algorithm* is much lower than that of *Recursive-efficiency algorithm*. This is mainly due to the advantages brought by the pre-upload of the $size = 0$ layer under merged layer with $LDA^* > 0$. For the VGG model, the $size = 0$ layer with $ELDA = -\infty$. From this, we know that under *Recursive-efficiency algorithm*, the $size = 0$ layer under merged layer with $LDA^* > 0$ will be uploaded in the last part of the DNN upload. Under *LMU algorithm*, we consider *Principle 2*. Based on this, we pre-upload $size = 0$ layer, and then upload the other layers of the merged layer to obtain a better *LDA*.

The most important point is that whether *IONN algorithm* or *Recursive-efficiency algorithm*, they are based on the shortest path method for DNN division, which is not suitable for the local-edge-cloud collaborative environment with multi-cloud/edge servers. However, *EosDNN* offloading scheme uses the *PSO-GA algorithm* to divide the DNN based on the local-edge-cloud collaborative environment to develop a migration plan and uses *LMU algorithm* to develop a detailed uploading plan, which is obviously more suitable for a multi-server environment.

D. Delay Under Different Computing Environments

In Fig. 15, we evaluate the migration Delay based on the *PSO-GA algorithm* in different computing environments, i.e., the local-edge environment (*L-E*), the local-cloud environment (*L-C*) and the local-edge-cloud collaborative environment (*L-E-C*).

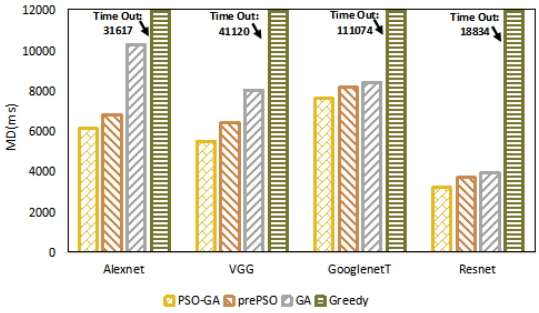


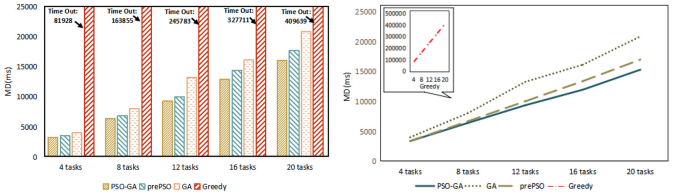
Fig. 16. Comparison of *MD*s of different algorithms under Alexnet, VGG, Googlenet and Resnet, each with 12 tasks.

It can be seen from Fig. 15(a) that the execution delay under the *L-E* environment is relatively higher, because the edge server has limited computing resources and insufficient computing power, resulting in waiting delay. With the help of scalable cloud servers, the execution time of large-scale tasks can be greatly reduced. In Fig. 15(b), the transmission delay in the *L-C* environment is much higher than that of the other computing environments, mainly because the cloud server is so far away from the client. This shows that migrating data-intensive tasks from the client to the cloud server is not always effective because it involves a large amount of data transfer. Considering that the edge server is close to the client, migrating some subtasks to the edge server can greatly reduce the transmission delay. In Fig. 15(c), migration Delay is the lowest under the *L-E-C* environment. The cloud computing center is too far away from clients, but its data transmission rate is low. The edge server has a high data transmission rate, but its computing power is insufficient to handle large-scale tasks. When considering characteristics of the edge and cloud, we adaptively choose to migrate some subtasks to the edge and cloud servers, which can effectively reduce the migration Delay.

E. Delay Performance Under Different Algorithms

From Fig. 16, the migration Delay based on *PSO-GA algorithm* is lower than *GA algorithm*, *perPSO algorithm* and the *Greedy algorithm*, whether in more complex models such as VGG, Googlenet, and ResNet or relatively simple models such as AlexNet. In addition, compared with *perPSO algorithm*, *PSO-GA algorithm* has a lower delay. This is because the compression layer has a larger amount of calculation and must be distributed on cloud servers with greater computing power. However, the transmission delay from the cloud server to the client is very high. Obviously, the migration plan generated by *PSO-GA algorithm* is significantly better than other algorithms and is more suitable for migrating tasks of different sizes. Compared with other algorithms, the delay performance is better, the algorithm performance is more stable, and the applicability is generally higher.

As shown in Fig. 17(b), the migration Delay under the four algorithms increases linearly with the increase of task amount. It is not difficult to see that with the increase of task amount, the migration Delay under *Greedy algorithm*, *prePSO algorithm* and *GA algorithm* have a steeper increase trend and more obvious fluctuation, among which *Greedy algorithm* has the largest fluctuation. However, the *PSO-GA algorithm* shows a



(a) Algorithm effect comparison (b) Increasing trend comparison

Fig. 17. The impact of the number of tasks on *MD* under different algorithms.

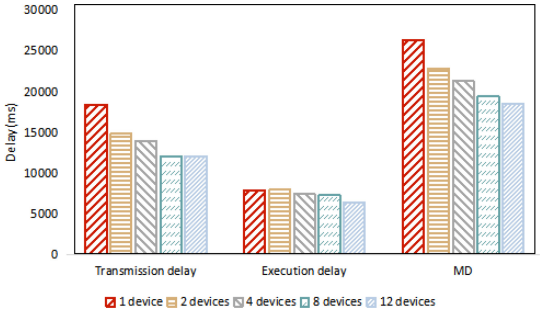


Fig. 18. The impact of the number of local devices on *MD*.

relatively stable increase trend. It can be observed from Fig. 17 that the *PSO-GA algorithm* has greater advantages in dealing with large-scale DNN tasks.

In Fig. 18, the migration Delay will gradually decrease as the number of local devices increases. We observe the relationship between migration delay, transmission delay, and execution delay, and find that as the number of clients increases, the reduction of migration delay is mainly reflected in the reduction of transmission delay. Obviously, the *PSO-GA algorithm* is suitable for large-scale DNN task migrating with multi-users.

VIII. Conclusion and Future Work

In this paper, we have addressed the joint optimization problem of DNN migration and DNN upload in a multi-user local-edge-cloud collaborative environment, where computing resources are too limited to support complex intelligent applications. We apply *PSO-GA algorithm* to obtain the distribution of the DNN layer that meets the minimum migration delay of multi-task parallelism and propose *LMU algorithm* to achieve more fine-grained DNN partitions and obtain better DNN query performance. It is verified that EosDNN is suitable for large-scale DNN parallelism in the multi-user local-edge-cloud collaborative environment with limited computing resources.

In future work, we will consider how to coordinate the balance between delay, energy consumption, and cost in real-world DNN migration. In addition, the unreasonable allocation of computing resources in the local-edge-cloud collaborative computing environment is also a direction that needs in-depth research.

References

[1] Z. Xu *et al.*, "Energy-aware inference offloading for DNN-driven applications in mobile edge clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 799–814, Apr. 2021.

[2] W. Niu *et al.*, "PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2020, pp. 907–922.

[3] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2018, pp. 401–411.

[4] H.-J. Lee, H.-J. Jeong, and S.-M. Moon, "Snapshot-based customization for offloading Web application computation in edge computing environment," to be published.

[5] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun. (MECOMM)*, 2018, pp. 31–36.

[6] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.

[7] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2019, pp. 1423–1431.

[8] H. Wang, G. Cai, Z. Huang, and F. Dong, "ADDA: Adaptive distributed DNN inference acceleration in edge computing environment," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2019, pp. 438–445.

[9] W. He, S. Guo, S. Guo, X. Qiu, and F. Qi, "Joint DNN partition deployment and resource allocation for delay-sensitive deep learning inference in IoT," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9241–9254, Oct. 2020.

[10] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM Sigplan Notices*, vol. 52, no. 1, pp. 615–629, 2017.

[11] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 73–82.

[12] M. Xu, Q. Zhou, H. Wu, W. Lin, K. Ye, and C. Xu, "PDMA: Probabilistic service migration approach for delay-aware and mobility-aware mobile edge computing," *Softw. Pract. Exp.*, to be published.

[13] G. Qu, H. Wu, R. Li, and P. Jiao, "DMRO: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 3448–3459, Sep. 2021.

[14] H. Wu, K. Wolter, P. Jiao, Y. Deng, Y. Zhao, and M. Xu, "EEDTO: An energy-efficient dynamic task offloading algorithm for blockchain-enabled IoT-edge-cloud orchestrated computing," *IEEE Internet Things J.*, vol. 8, no. 4, pp. 2163–2176, Feb. 2021.

[15] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.

[16] Y. Huang, B. Lin, Y. Zheng, J. Hu, Y. Mo, and X. Chen, "Cost efficient offloading strategy for DNN-based applications in edge-cloud environment," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl. Big Data Cloud Comput. Sustain. Comput. Commun. Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, Dec. 2019, pp. 331–337.

[17] D. Liu, X. Chen, Z. Zhou, and Q. Ling, "HierTrain: Fast hierarchical edge AI learning with hybrid parallelism in mobile-edge-cloud computing," *IEEE Open J. Commun. Soc.*, vol. 1, pp. 634–645, 2020.

[18] H. Wu, W. J. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1464–1480, Jul. 2019.

[19] I. A. Elgendy, W.-Z. Zhang, Y. Zeng, H. He, Y.-C. Tian, and Y. Yang, "Efficient and secure multi-user multi-task computation offloading for mobile-edge computing in mobile IoT networks," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 4, pp. 2410–2422, Dec. 2020.

[20] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1657–1681, 3rd Quart., 2017.

[21] H. Liu, H. Zheng, M. Jiao, and G. Chi, "SCADS: Simultaneous computing and distribution strategy for task offloading in mobile-edge computing system," in *Proc. IEEE Int. Conf. Commun. Technol.*, 2018, pp. 1286–1290.

[22] W. Zhang, B. Zhou, W. Dang, and S. Hu, "A lightweight energy-efficient computational offloading scheme in mobile edge computing," in *Proc. 11th ACM Int. Conf. Future Energy Syst.*, Jun. 2020, pp. 560–565.

[23] L. Cui, J. Zhang, L. Yue, Y. Shi, H. Li, and D. Yuan, "A genetic algorithm based data replica placement strategy for scientific applications in clouds," *IEEE Trans. Services Comput.*, vol. 11, no. 4, pp. 727–739, Jul./Aug. 2018.

[24] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proc. 24th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, 2010, pp. 400–407.

[25] M. Masdari, F. Salehi, M. Jalali, and M. Bidaki, "A survey of PSO-based scheduling algorithms in cloud computing," *J. Netw. Syst. Manage.*, vol. 25, no. 1, pp. 122–158, 2016.

[26] M. Deng, H. Tian, and B. Fan, "Fine-granularity based application offloading policy in cloud-enhanced small cell networks," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC)*, May 2016, pp. 638–643.

[27] B. Lin *et al.*, "A time-driven data placement strategy for a scientific workflow combining edge computing and cloud computing," *IEEE Trans. Ind. Informat.*, vol. 15, no. 7, pp. 4254–4265, Jul. 2019.

[28] S. Huang, J. B. Yang, and H. J. Ding, "Research on GA-DPSO virtual machine scheduling algorithm based on comprehensive utilization of host resource," *Adv. Mater. Res.*, vols. 791–793, pp. 1373–1376, Sep. 2013.

[29] W. Xu and S. Guo, "A multi-objective and multi-dimensional optimization scheduling method using a hybrid evolutionary algorithms with a sectional encoding mode," *Sustainability*, vol. 11, no. 5, p. 1329, 2019.

[30] B. Lin, Y. Huang, J. Zhang, J. Hu, X. Chen, and J. Li, "Cost-driven off-loading for DNN-based applications over cloud, edge, and end devices," *IEEE Trans. Ind. Informat.*, vol. 16, no. 8, pp. 5456–5466, Aug. 2020.

[31] I. Jeong, H.-J. Jeong, and S.-M. Moon, "Snapshot-based offloading for machine learning Web app: Work-in-progress," in *Proc. 13th ACM Int. Conf. Embedded Softw. Compan. (EMSOFT)*, 2017, pp. 1–2.

[32] K. Y. Shin, H.-J. Jeong, and S.-M. Moon, "Enhanced partitioning of DNN layers for uploading from mobile devices to edge servers," in *Proc. 3rd Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2019, pp. 35–40.

[33] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Trans. Mobile Comput.*, vol. 20, no. 2, pp. 565–576, Feb. 2021.

[34] H.-J. Jeong, "Lightweight offloading system for mobile edge computing," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2019, pp. 451–452.

[35] L. Lockhart, P. Harvey, P. Imai, P. Willis, and B. Varghese, "Scission: Performance-driven and context-aware cloud-edge distribution of deep neural networks," in *Proc. IEEE/ACM 13th Int. Conf. Utility Cloud Computing (UCC)*, Dec. 2020, pp. 257–268.

[36] X. Tian, J. Zhu, T. Xu, and Y. Li, "Mobility-included DNN partition offloading from mobile devices to edge clouds," *Sensors*, vol. 21, no. 1, p. 229, 2021.

[37] Hochba and S. Dorit, "Approximation algorithms for NP-hard problems," *ACM SIGACT News*, vol. 28, no. 2, pp. 40–52, 1997.

[38] H. Lu, X. He, M. Du, X. Ruan, Y. Sun, and K. Wang, "Edge QoE: Computation offloading with deep reinforcement learning for Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 9255–9265, Oct. 2020.

[39] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu, "iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 7011–7024, Aug. 2019.

[40] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *Proc. IEEE Int. Conf. Evol. Comput. IEEE World Congr. Comput. Intell.*, 1998, pp. 69–73.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

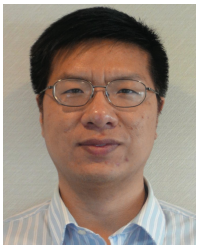
[43] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[45] P. Zhao, H. Tian, and B. Fan, "Partial critical path based greedy offloading in small cell cloud," in *Proc. IEEE 84th Veh. Technol. Conf. (VTC-Fall)*, Sep. 2016, pp. 1–5.

Min Xue received the bachelor's degree from Qingdao University of Science and Technology, Qingdao, China, in 2018. She is currently pursuing the master's degree with the Center for Applied Mathematics, Tianjin University, China. Her research interests include deep learning, deep reinforcement learning, cloud computing, and mobile edge computing.

Huaming Wu (Member, IEEE) received the B.E. and M.S. degrees in electrical engineering from Harbin Institute of Technology, China, in 2009 and 2011, respectively, and the Ph.D. degree (Highest Hons.) in computer science from Freie Universität Berlin, Germany, in 2015. He is currently an Associate Professor with the Center for Applied Mathematics, Tianjin University, China. His research interests include wireless networks, mobile edge computing, Internet of Things, and deep learning.



Ruidong Li (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the University of Tsukuba in 2005 and 2008, respectively. He was a Senior Researcher with the National Institute of Information and Communications Technology, Japan. He is an Associate Professor with Kanazawa University, Japan. His research interests include future networks, big data, intelligent Internet edge, Internet of Things, network security, information-centric network, artificial intelligence, quantum Internet, cyber-physical system, and wireless networks. He serves as the Secretary of IEEE ComSoc Internet Technical Committee, and are the founders and chairs of IEEE SIG on Big Data Intelligent Networking and IEEE SIG on Intelligent Internet Edge. He is an Associate Editor of IEEE INTERNET OF THINGS JOURNAL, and also served as the Guest Editor for a set of prestigious magazines, transactions, and journals, such as *IEEE Communications Magazine*, IEEE NETWORK, and IEEE TRANSACTIONS ON NETWORK SCIENCE AND ENGINEERING. He also served as chairs for several conferences and workshops, such as the General Co-Chair for IEEE MSN 2021, AIVR2019, and IEEE INFOCOM 2019/2020/2021 ICCN Workshop, and a TPC Co-Chair for IWQoS 2021, IEEE MSN 2020, BRAINS 2020, IEEE ICDCS 2019/2020 NMIC Workshop, and ICCSSE 2019. He is a Senior Member of IEICE.

Minxian Xu (Member, IEEE) received the B.Sc. and M.Sc. degrees in software engineering from the University of Electronic Science and Technology of China in 2012 and 2015, respectively, and the Ph.D. degree from the University of Melbourne in 2019. He is currently an Assistant Professor with Shenzhen the Institutes of Advanced Technology, Chinese Academy of Sciences. He has coauthored over 20 peer-reviewed papers published in prominent international journals and conferences, such as *ACM Computing Surveys*, IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING, IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING, *Journal of Systems and Software*, *Journal of Parallel and Distributed Computing*, ICSOC, and IEEE INTERNET OF THINGS JOURNAL. His research interests include resource scheduling and optimization in cloud computing. His Ph.D. Thesis was awarded the 2019 IEEE TCSC Outstanding Ph.D. Dissertation Award. He is member of CCF.

Pengfei Jiao received the Ph.D. degree in computer science from Tianjin University, Tianjin, China, in 2018, where he is a Lecturer with the Center of Biosafety Research and Strategy. He has published more than 50 international journals and conference papers. His current research interests include complex network analysis, data mining and graph neural network, and currently working on temporal community detection, link predication, network embedding, recommender systems, and applications of statistical network model.