Fast AI Model Partition for Split Learning Over Edge Networks

Zuguang Li, *Graduate Student Member, IEEE*, Wen Wu, *Senior Member, IEEE*, Shaohua Wu, *Senior Member, IEEE*, and Xuemin Shen, *Fellow, IEEE*

***Abstract*—Split learning (SL) is a distributed learning paradigm that can enable computation-intensive artificial intelligence (AI) applications by partitioning AI models between mobile devices and edge servers. However, the model partitioning problem in SL becomes challenging due to the diverse and complex architectures of AI models. In this paper, we formulate an optimal model partitioning problem to minimize training delay in SL. To solve the problem, we represent an arbitrary AI model as a directed acyclic graph (DAG), where the model's layers and inter-layer connections are mapped to vertices and edges, and training delays are captured as edge weights. Then, we propose a general model partitioning algorithm by transforming the problem into a minimum *s-t* cut problem on the DAG. The theoretical analysis shows that these two problems are equivalent, such that the optimal model partition can be obtained via a maximum-flow method. Furthermore, taking AI models with block structures into consideration, we design a low-complexity block-wise model partitioning algorithm to determine the optimal model partition. Specifically, the algorithm simplifies the DAG by abstracting each block (i.e., a repeating component comprising multiple layers in an AI model) into a single vertex. Extensive experimental results on a hardware testbed equipped with NVIDIA Jetson devices demonstrate that the proposed solution can reduce algorithm running time by up to 13× and training delay by up to 38.95%, compared to state-of-the-art baselines.**

***Index Terms*—Split learning, fast model partition, block-wise, directed acyclic graph.**

I. Introduction

With the large-scale deployment of Internet of Things (IoT) devices and the advancement of sensing technologies, massive amounts of data are continuously generated by mobile devices in edge networks. Leveraging such data, artificial intelligence (AI) can enable a wide range of intelligent applications in edge networks, including healthcare, autonomous driving, and smart cities [1]. Due to data privacy concerns, training AI models using traditional centralized learning is often difficult [2]. Distributed learning frameworks, e.g., federated learning (FL), have been proposed to support collaborative model training without sharing raw data [3]. However, FL requires training the entire model on mobile devices, which is challenging, especially when their computing capabilities are limited.

Split learning (SL), as a promising distributed learning paradigm, has been proposed to address the weakness of FL [4]. Specifically, SL divides the entire AI model into a device-side model and a server-side model. Each device trains only its device-side model using its own data and transmits intermediate results (i.e., smashed data and gradients) to the edge server instead of raw data [5]. In this way, SL can offload computational load from devices to the edge server while allowing devices to retain their raw data. In the SL framework, model partitioning is essential, particularly for edge networks, as it directly affects both communication overhead and device-side computational load.

In the literature, several model partitioning methods have been proposed for SL to accelerate AI model training. In general, AI models can be classified into linear and non-linear architectures according to their layer structures [6]. Linear AI models follow a strictly sequential structure, where each layer depends only on the output of the previous layer (e.g., LeNet [7] and AlexNet [8]). In contrast, non-linear AI models contain complex connections such as skip connections or parallel branches (e.g., residual blocks in ResNet [9]), which create multiple dependency paths during computation. For linear AI models, the number of available cuts is limited, and a brute-force search is a widely adopted approach to determine the optimal cut that minimizes model training latency [10], [11]. For non-linear AI models, complex structures make brute-force search impractical for finding the optimal cut due to its prohibitive computational complexity. To determine the cut in non-linear AI models, a non-linear module was integrated into a single block, thereby transforming the non-linear model into a simplified linear structure for partitioning [6]. A regression-based method was designed to quantify the relationship between the cut and both the computing delay and communication delay, enabling the optimal cut to be obtained by solving an optimization problem [12]. In addition, a general-structure partitioning method was proposed to handle non-linear

Received 29 December 2025; revised 30 April 2026; accepted 7 June 2026. Date of publication 1 July 2026; date of current version 5 October 2026. This work was supported by the Pengcheng Laboratory Major Key Project under Grant PCL2025A08 and 2025QYA002. Recommended for acceptance by W. Yuan. *(Corresponding author: Wen Wu.)*

Zuguang Li and Wen Wu are with the School of Electronics and Information Engineering, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China, and also with the Department of Strategic and Advanced Interdisciplinary Research, Pengcheng Laboratory, Shenzhen 518055, China (e-mail: lizuguang@stu.hit.edu.cn; wuw02@pcl.ac.cn).

Shaohua Wu is with the Guangdong Provincial Key Laboratory of Aerospace Communication and Networking Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China, and also with the Department of Broadband Communication, Pengcheng Laboratory, Shenzhen 518055, China (e-mail: hitwush@hit.edu.cn).

Xuemin Shen is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: sshen@uwaterloo.ca).

This article has supplementary downloadable material available at https://doi.org/10.1109/TMC.2026.3709072, provided by the authors.

Digital Object Identifier 10.1109/TMC.2026.3709072

1536-1233 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

AI models, which decomposes the AI model into multiple parallel linear paths and applies binary search to determine the optimal partition for each path [13]. However, these approaches are generally suboptimal for non-linear AI models. Determining the optimal cut for arbitrary AI models faces three primary challenges: (*i*) AI models are highly complex, and accurately representing the relationship among layers is challenging; (*ii*) during model training, device-side computation latency, server-side computation latency, and communication latency should be associated with the corresponding layers; and (*iii*) an efficient method is required to quickly determine the optimal cut without incurring excessive computational latency. Therefore, can we develop an efficient and generalizable approach to determine the optimal model partition for arbitrary AI models?

In this paper, we investigate the model partitioning problem for SL, aiming to quickly determine the optimal model partition that minimizes training delay in edge networks. To achieve this goal, *firstly*, we represent an AI model as a directed acyclic graph (DAG), where the AI model's layers and inter-layer connections are mapped to vertices and edges, respectively. To encode the computation and transmission delays in the DAG, we assign three types of edge weights: *(i) device execution weight*, defined as the sum of the computation delay of training the corresponding layer on the device and the transmission delay for uploading that layer's parameters; *(ii) server execution weight*, defined as the sum of the computation delay of training the corresponding layer on the server and the transmission delay for downloading that layer's parameters; and *(iii) propagation weight*, defined as the transmission delay of the smashed data and the corresponding gradients propagated between two connected layers. *Secondly*, for an arbitrary AI model, we propose a general model partitioning algorithm to determine the optimal cut. Specifically, the optimal model partitioning problem is reformulated as a minimum *s–t* cut problem on the DAG. We further prove that these two problems are equivalent via rigorous theoretical analysis, meaning that solving the minimum *s–t* cut directly yields the optimal model partition. Based on this equivalence, the optimal cut can be efficiently obtained using classical maximum-flow algorithms. *Thirdly*, for an AI model with block structures, we propose a low-complexity block-wise model partitioning algorithm to determine the optimal cut. Specifically, the algorithm simplifies the DAG by abstracting each recurrent block (i.e., a module consisting of multiple layers) into a single vertex. In this way, the computational complexity of searching for the optimal cut can be further reduced. Our experimental results on a hardware testbed with NVIDIA Jetson devices demonstrate that the proposed solution can: *(i)* determine the optimal model partition for SL within milliseconds, where the block-wise model partitioning algorithm achieves up to a $13\times$ reduction in running time compared with the general model partitioning algorithm; and *(ii)* accelerate model training by reducing the overall training delay by up to 38.95% compared with the regression-based method. The main contributions of this paper can be summarized as follows:

- We represent an arbitrary AI model as a general DAG and formulate an optimal model partitioning problem to minimize the training delay in SL;
- We propose a general model partitioning algorithm to determine the optimal model partition, which equivalently transforms the optimal model partitioning problem into a minimum *s-t* cut problem;
- We propose a block-wise model partitioning algorithm for AI models, which can quickly identify the optimal model partition.

The remainder of this paper is organized as follows. Section II reviews the related works, followed by the system model in Section III. The DAG-based representation of AI models is presented in Section IV. The general and block-wise model partitioning algorithms are presented in Sections V and VI, respectively. Performance evaluation is provided in Section VII. Finally, Section VIII concludes this paper.

II. Related Work

The SL paradigm has attracted widespread attention and investigation due to its ability to enable flexible computing load sharing across devices and edge servers for collaborative training. In the 3 rd generation partnership project (3GPP) Release 18, the performance advantages of AI model partition have been recognized, and protocol-level primitives have been explored to support model partition between end devices and the server [14]. A communication-aware SL framework with an early-exit mechanism was developed to accommodate devices with diverse computational capabilities and fluctuating channel conditions [15]. An online model partition strategy was proposed to minimize energy and latency costs under dynamic wireless channel conditions [16]. To support dynamic networks, a cluster-based outgoing SL framework was introduced, enabling a first-outgoing-then-sequential training paradigm for heterogeneous devices [17]. Similarly, a parallel SL framework was proposed where all devices collaboratively train the device-side model with the server, addressing computational heterogeneity in edge networks [10]. To further improve communication efficiency, a split federated learning framework was designed, where cut selection and bandwidth allocation are jointly optimized to minimize overall system latency [18]. Existing work has explored SL to address challenges such as heterogeneous computing capabilities and dynamic channel conditions in edge networks. In contrast, our work focuses on fast AI model partition for SL to reduce the computational complexity and the overall training delay.

The selection of the cut layer in SL directly affects computational load and communication overload. A theoretical analysis of split federated learning was conducted to examine how the cut layer selection influences convergence and overall performance [19]. Another study modeled the cut layer selection process as a Stackelberg game to optimize client incentives and data contributions while balancing privacy preservation and energy consumption [20]. To determine the optimal cut layer, various approaches have been proposed, including brute-force search [10], binary search [13], regression-based search [21], and graph-based search approaches [22], [23], [24]. A brute-force search method was proposed to identify the optimal cut layer for either minimum latency or mobile energy consumption, based on profiling per-layer execution time and energy usage across different AI models [25]. A binary search-based partition algorithm was designed for pipeline inference in line-structured models [13]. In graph-based approaches, a

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

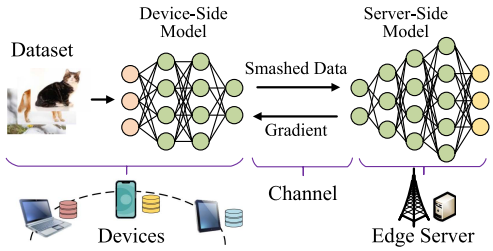


Fig. 1. Considered wireless SL scenario in edge networks.

DAG-based method was proposed to select the optimal cut layer under dynamic network conditions, thereby reducing inference latency in edge–cloud systems [22]. Another approach addressed multi-partitioning in DAG-topology networks through graph transformation and partitioning techniques [24]. The existing model partition approaches are primarily designed for linear neural networks, limiting their applicability to more complex non-linear neural networks. Differently, our work focuses on finding the optimal model partition for arbitrary AI models.

III. System Model

A. Considered Scenario

In this subsection, we present a typical SL scenario in an edge network and the model training process. As shown in Fig. 1, the system includes mobile devices and an edge server. The AI model is partitioned between the device and the server, with intermediate data transmitted via wireless channels. Key system components are described as follows:

- *Edge server:* The edge server is equipped with a base station that can perform server-side model training. Moreover, it is responsible for collecting network information, such as device computing capabilities and channel conditions, to support decision-making for the model partition.
- *Mobile Devices:* The devices are endowed with computing capabilities, which can perform device-side model training. However, their computing capabilities are significantly weaker than the edge server's, which motivates offloading part of the training workload to the server to accelerate the overall training process. They have local datasets and corresponding labels for model training.

Considering the heterogeneous device capabilities and the dynamic channel conditions in edge networks, the server dynamically determines the optimal model partition for each device at each training epoch. Before training begins, the model parameters are randomly initialized. In each training epoch, the server collects device and network information and selects a device for training in a round-robin manner. Based on the information, the server determines the model partition, denoted by $c = \{\mathcal{V}_D, \mathcal{V}_S\}$, where $\mathcal{V}_D$ and $\mathcal{V}_S$ denote the sets of layers assigned to the device and server, respectively. The device and server then jointly train the AI model over N_{loc} local iterations.

During each local iteration, the device performs forward propagation on the device-side model and transmits the smashed data and corresponding labels to the server. The server completes forward and backward propagation on the server-side model and returns the gradients to the device, which subsequently updates its device-side model. After completing local iterations, the device uploads its updated device-side model to the server, which integrates the updates and distributes the updated device-side model to the next selected device. This process is repeated across devices over multiple training epochs until the global model converges.

B. Training Delay

In SL, the computation and transmission delays are analyzed as follows.

1) Computation Delay: During each training epoch, the device-side model and the server-side model are trained at the device and server, respectively, whose delays are analyzed as follows:

- *Device-side model computation delay:* The computation delay in processing layer v_i is defined as ξ_{D,v_i}. Note that ξ_{D,v_i} consists of the forward and backward propagation computation delay at layer v_i. The device-side model computation delay can be defined as

$$T_{D,C} = \sum_{v_i \in \mathcal{V}_D} \xi_{D,v_i}. \tag{1}$$

- *Server-side model computation delay:* For the server, the computation delay in processing layer v_i is defined as ξ_{S,v_i}. Hence, its computation delay can be defined as

$$T_{S,C} = \sum_{v_i \in \mathcal{V}_S} \xi_{S,v_i}. \tag{2}$$

2) Transmission Delay: During SL, the device-side model, labels, smashed data, and its gradients are exchanged between the device and server, incurring transmission delay. Since the transmission delay of labels is independent of the cut layer selection, it can be ignored. The analysis of other transmission delays is as follows.

- *Device-side model downloading delay:* Let R_S denote the transmission rate from the server to the device, and k_{v_i} denote the data size of the parameters in layer v_i. Hence, the downloading delay for transmitting the device-side model can be defined as

$$T_{S,D} = \frac{\sum_{v_i \in \mathcal{V}_D} k_{v_i}}{R_S}. \tag{3}$$

- *Smashed data and transmission delay:* Let R_D denote the average transmission rate from the device to the server, and a_{v_i} denote the data size of the smashed data of vertex v_i in the forward propagation. The transmission delay for transmitting the smashed data can be defined as

$$T_{D,S} = \frac{\sum_{v_i \in \mathcal{V}_c} a_{v_i}}{R_D}, \tag{4}$$

where $\mathcal{V}_c$ is the set of end layers on the device-side model after the cut splits the AI model.

- *Smashed data's gradient transmission delay:* Let $\tilde{a}_{v_i}$ denote the size of the smashed data's gradient received by vertex v_i during backward propagation. During AI model

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

training, the size of smashed data is equal to that of its corresponding gradient, i.e., $\tilde{a}_{v_i} = a_{v_i}$. The transmission delay for transmitting the gradient can be defined as

$$T_{S,G} = \frac{\sum_{v_i \in \mathcal{V}_c} \tilde{a}_{v_i}}{R_S}. \tag{5}$$

- *Device-side model uploading delay:* The device-side model uploading delay can be defined as

$$T_{D,U} = \frac{\sum_{v_i \in \mathcal{V}_D} k_{v_i}}{R_D}. \tag{6}$$

3) Overall Model Training Delay: Taking all the computation and transmission delay components into account, the model training delay is given by

$$T(c) = N_{loc}\left(T_{D,C} + T_{D,S} + T_{D,B} + T_{S,C} + T_{S,G}\right) + T_{D,U} + T_{S,D}. \tag{7}$$

Our goal is to find the optimal model partition that minimizes the overall training delay, i.e., $\min T(c)$. However, solving this optimization problem is challenging, especially when the structure of an AI model is complex and non-linear. Linear AI models typically follow a sequential structure in which layers are executed one after another, e.g., LeNet and AlexNet. For linear AI models, the number of available cuts is limited, and the optimal cut can be efficiently obtained through a simple brute-force search. In contrast, non-linear AI models contain complex computational dependencies such as residual connections or parallel branches, where multiple paths may exist between layers (e.g., residual blocks in ResNet). These complex dependencies make it difficult to directly determine the optimal cut due to the large number of possible cuts and intricate data dependencies.

IV. DAG-Based Representation of AI Models

In this section, we represent an AI model as a DAG, with edge weights encoding the computation and transmission delays. Based on this DAG, we formulate the optimal model partition problem as a minimum *s-t* cut problem.

A. DAG Model

To find the optimal model partition, a unified and structured representation of AI models is required. Since an AI model inherently consists of layers with computational and data dependencies, it can naturally be abstracted as a DAG, where vertices represent layers and edges capture their dependencies. The acyclic property ensures that the execution order strictly follows the intrinsic computational logic of the AI model. Moreover, this representation enables us to explicitly encode computation and communication delays and facilitates the use of efficient graph-based optimization techniques, such as minimum *s-t* cut algorithms, for optimal model partitioning. The construction of the DAG is described as follows.

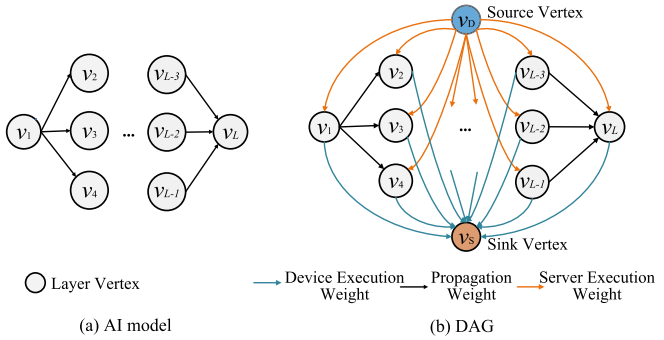


Fig. 2. Illustration of representing an AI model as a DAG. (a) The AI model consists of multiple layers. (b) The corresponding DAG adds a source and a sink vertex.

1) DAG Construction: As illustrated in Fig. 2(a), a typical AI model consists of various types of layers, such as convolutional layers, pooling layers, and fully connected layers. These layers are connected via data dependencies, forming a computational graph. This graph can be represented as a DAG $\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A)$, where $\mathcal{V}_A = \{v_1, v_2, \ldots, v_L\}$ denotes the set of layer vertices. Each vertex v_i corresponds to a specific layer and L is the total number of layers in the AI model. Let $\mathcal{E}_A$ be the set of directed edges, and each edge $(v_i, v_j) \in \mathcal{E}_A$ indicates that the output of layer v_i is used as the input to layer v_j in the AI model. Therefore, the AI model structure can be represented as a DAG.

To represent the computation flow from device to server, we introduce two virtual vertices: v_D, *source vertex*, denoting a virtual device, and v_S, *sink vertex*, denoting a virtual server. In addition, we add edges from the source vertex v_D to each layer vertex $v_i \in \mathcal{V}$, and from each layer vertex $v_i \in \mathcal{V}$ to the sink vertex v_S, so that the entire AI model is enclosed between the source and sink vertices, as shown in Fig. 2(b). The DAG can be defined as

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \tag{8}$$

where $\mathcal{V} = \mathcal{V}_A \cup \{v_D, v_S\}$ and $\mathcal{E} = \mathcal{E}_A \cup \{(v_D, v_i)\}_{v_i \in \mathcal{V}_A} \cup \{(v_i, v_S)\}_{v_i \in \mathcal{V}_A}$. Note that $\{(v_D, v_i)\}_{v_i \in \mathcal{V}_A}$ denotes the set of all directed edges from v_D to each vertex $v_i \in \mathcal{V}_A$, and similarly, $\{(v_i, v_S)\}_{v_i \in \mathcal{V}_A}$ denotes the set of all directed edges from $v_i \in \mathcal{V}_A$ to the sink v_S.

2) Mapping Delay into DAG: The main challenge in using DAG to solve the optimal model partitioning problem is that it cannot directly represent computation delay, device-side model uploading delay, or device-side model downloading delay via vertex or edge weights. This is because the computation delay of each layer depends on whether it is executed on the device or the server. In contrast, the uploading and downloading delays depend on the specific subset of layers trained on the device. To address this, the model training delays (including the computation delay and transmission delay) are mapped into three types of edge weights in the DAG as follows:

- *Device execution weight:* The weight of the directed edge from each layer vertex $v_i \in \mathcal{V}$ to the sink vertex v_S represents the device execution weight when v_i is executed

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: DAG Building Process.

Input: $\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A)$;
Output: $\mathcal{G}$;
$\mathcal{V} = \mathcal{V}_A$, $\mathcal{E} = \mathcal{E}_A$;
for *each* $v_i \in \mathcal{V}$ **do**
  Add edges (v_D, v_i) and (v_i, v_S) to $\mathcal{E}$;
end
for *each* $(v_i, v_j) \in \mathcal{E}$ **do**
  if $v_i == v_D$ **then**
    Calculate $w_{(v_i,v_j)}$ based on Eq. (9);
  else if $v_j == v_S$ **then**
    Calculate $w_{(v_i,v_j)}$ based on Eq. (10);
  else
    Calculate $w_{(v_i,v_j)}$ based on Eq. (11);
  end
  Assign $w_{(v_i,v_j)}$ to edge (v_i, v_j);
end
$\mathcal{G} = (\mathcal{V} \cup \{v_D, v_S\}, \mathcal{E})$.

on the device. This weight is defined as the sum of the computation delay for training v_i on the device and the transmission delay for uploading v_i during the device-side model update. It is calculated as

$$w_{(v_i,v_S)} = N_{loc}\xi_{D,v_i} + \frac{k_{v_i}}{R_D}. \tag{9}$$

- *Server execution weight:* The weight of the directed edge from source vertex v_D to each layer vertex $v_i \in \mathcal{V}$ represents the server execution weight when v_i is executed on the server. This weight is defined as the sum of the computation delay for training v_i on the server and the transmission delay for downloading v_i during the device-side model distribution. It is calculated as

$$w_{(v_D,v_i)} = N_{loc}\xi_{S,v_i} + \frac{k_{v_i}}{R_S}. \tag{10}$$

- *Propagation weight:* The weight of the directed edge between any two layer vertices $v_i, v_j \in \mathcal{V}$ represents the propagation weight. This weight is defined as the sum of the transmission delay of v_i's smashed data during forward propagation and v_j's gradient during backward propagation. It is calculated as

$$w_{(v_i,v_j)} = N_{loc}\left(\frac{a_{v_i}}{R_D} + \frac{\tilde{a}_{v_i}}{R_S}\right). \tag{11}$$

Here, $\tilde{a}_{v_i}$ denotes the size of the gradient received by vertex v_i from v_j during backward propagation, which equals the size of the smashed data in forward propagation, i.e., $\tilde{a}_{v_i} = a_{v_i}$.

Through the above process, the computation and transmission delays are encoded as the weights of the edges in DAG $\mathcal{G}$. As shown in Fig. 2(b), the edges with device execution, server execution, and propagation weights are shown in blue, orange, and black, respectively. The detailed procedure for constructing the DAG from an AI model is presented in Algorithm 1. Note that in the DAG, vertices are unweighted and represent only the layers of the AI model, as well as the virtual device and the virtual server.

B. Problem Formulation

We aim to minimize the overall training delay in SL, which can be reformulated as a minimum *s-t* cut problem in the DAG, i.e., finding an *s-t* cut that minimizes the total weight of the edges intersected by the cut. An *s-t* cut partitions the vertex set into two disjoint subsets $\mathcal{V}_D$ and $\mathcal{V}_S$, such that the source vertex $v_D \in \mathcal{V}_D$ and the sink vertex $v_S \in \mathcal{V}_S$. The value of an *s-t* cut $c = \{\mathcal{V}_D, \mathcal{V}_S\}$ is defined as the sum of the weights of all edges directed from $\mathcal{V}_D$ to $\mathcal{V}_S$. Let $\mathcal{W}(c)$ denote the set of weights of these edges, then the value of the cut can be expressed as $T_{\mathcal{G}}(c) = \sum_{w \in \mathcal{W}(c)} w$. Thus, the minimum *s-t* cut problem in the DAG can be formulated as follows:

$$\begin{aligned} &\min_{c} T_{\mathcal{G}}(c) \\ &\text{s.t. } c \in \mathcal{C}, \\ &\quad v_D \in \mathcal{V}_D, \\ &\quad v_S \in \mathcal{V}_S, \\ &\quad v_i \in \mathcal{V}_S \Rightarrow v_j \in \mathcal{V}_S, \forall (v_i, v_j) \in \mathcal{E}. \end{aligned} \tag{12}$$

Here, $\mathcal{C}$ is a set of all available cuts in the DAG, and the first constraint guarantees the feasibility of the cut selection. The second and third constraints ensure that the source vertex is placed on the device, while the sink vertex is placed on the server, respectively. The last constraint ensures that no vertex in $\mathcal{V}_S$ is the parent of any vertex in $\mathcal{V}_D$. This means that it is not possible to obtain an optimal cut, as device-side computations must wait for server-side computations to finish during forward propagation.

This problem cannot be directly solved using standard minimum *s-t* cut methods. In the DAG, if the cut intersects multiple outgoing edges from the same parent vertex, the corresponding propagation weight may be counted multiple times, leading to an overestimation of the cut value. However, in the AI model, the smashed data and its corresponding gradient from a parent vertex only need to be transmitted once, regardless of how many outgoing edges it has. To address this issue, we introduce a general model partitioning algorithm in the following section.

V. General Model Partitioning Algorithm

In this section, we propose a general model partitioning algorithm to determine the optimal model partition for arbitrary AI model architectures. We also prove that the cut obtained by this algorithm is optimal.

A. Algorithm Design

To eliminate the overestimation of propagation weights in the DAG, we propose a general model partitioning algorithm. This algorithm restructures the DAG to prevent duplicate transmission-weight counts from the same parent vertex. The main steps are described as follows.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

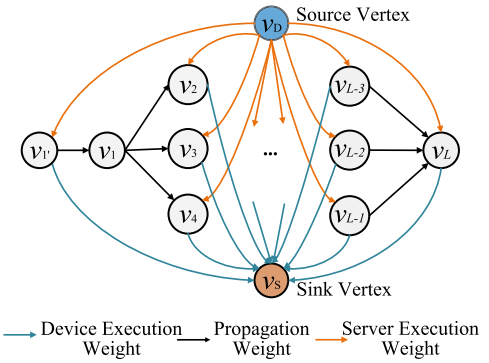


Fig. 3. Illustration of adding auxiliary vertices (e.g., $v_{1'}$) for parent vertices that have multiple child vertices.

1) Initialize a DAG $\mathcal{G}' = (\mathcal{V}' \cup \{v_D, v_S\}, \mathcal{E}')$ by setting $\mathcal{V}' = \mathcal{V}$ and $\mathcal{E}' = \mathcal{E}$. Then, identify the set of parent vertices that have multiple child vertices in $\mathcal{G}'$, denoted as $\mathcal{V}_\mathcal{P}$.
2) For each parent vertex $v_p \in \mathcal{V}_\mathcal{P}$, an auxiliary vertex $v_{p'}$ is introduced into the DAG, as illustrated in Fig. 3.
3) All incoming edges originally connected to v_p are redirected to the new vertex $v_{p'}$, while preserving their original weights:

$$w_{(v_i, v_{p'})} = w_{(v_i, v_p)}, \tag{13}$$

where v_i is a parent vertex of v_p. For example, in Fig. 3, edge (v_D, v_1) becomes $(v_D, v_{1'})$, satisfying $w_{(v_D, v_{1'})} = w_{(v_D, v_1)}$.
4) The edge from v_p to the sink vertex v_S is also redirected to originate from $v_{p'}$, with its weight unchanged:

$$w_{(v_{p'}, v_S)} = w_{(v_p, v_S)}. \tag{14}$$

As shown in Fig. 3, edge (v_1, v_S) becomes $(v_{1'}, v_S)$, and we have $w_{(v_{1'}, v_S)} = w_{(v_1, v_S)}$.
5) Add a new edge from the auxiliary vertex to the original parent vertex, i.e., $(v_{p'}, v_p)$, with a weight equal to the propagation cost from v_p to one of its child vertices:

$$w_{(v_{p'}, v_p)} = w_{(v_p, v_j)}, \tag{15}$$

where v_j is a child of v_p. For example, in Fig. 3, edge $(v_{1'}, v_1)$ is added with $w_{(v_{1'}, v_1)} = w_{(v_1, v_2)}$.

After the above process, we build a new DAG $\mathcal{G}'$ for non-linear neural networks. This new DAG resolves the issue of counting the same propagation weight multiple times when a cut intersects multiple edges from the same parent vertex. The optimal cut can then be obtained by solving the minimum *s-t* cut problem on $\mathcal{G}'$, which can be efficiently solved using standard maximum-flow algorithms, such as Dinic's algorithm [26]. For linear neural networks, this issue does not exist, as each layer has at most one child. Thus, the optimal cut can be directly computed on the original DAG $\mathcal{G}$ using simple algorithms such as the brute-force search method. The detailed procedure of the general model partitioning algorithm is presented in Algorithm 2.

Algorithm 2: General Model Partition Algorithm.

Input: $\mathcal{G} = (\mathcal{V} \cup \{v_D, v_S\}, \mathcal{E})$;
Output: c^*;
Find out the set $\mathcal{V}_\mathcal{P}$ of parent vertices with several child vertices;
if $\mathcal{V}_\mathcal{P} = \emptyset$ **then**
  Obtain minimum *s-t* cut $c^*_\mathcal{G}$ of $\mathcal{G}$ via the brute-force search;
  $c^* = c^*_\mathcal{G}$;
else
  $\mathcal{E}' \leftarrow \mathcal{E}$, $\mathcal{V}' \leftarrow \mathcal{V}$;
  for v_p *in* $\mathcal{V}_\mathcal{P}$ **do**
    Add auxiliary vertex $v_{p'}$ to $\mathcal{V}'$;
    The edges pointing to parent vertex v_p are migrated from v_p to $v_{p'}$;
    Calculate $w_{(v_i, v_{p'})}$ based on Eq. (13);
    Assign $w_{(v_i, v_{p'})}$ to edge $(v_i, v_{p'})$ in $\mathcal{G}'$;
    The edge between v_p and v_S are migrated from v_p to $v_{p'}$;
    Calculate $w_{(v_{p'}, v_S)}$ based on Eq. (14);
    Assign $w_{(v_{p'}, v_S)}$ to edge $(v_{p'}, v_S)$ in $\mathcal{G}'$;
    Add an edge $(v_{p'}, v_p)$ to $\mathcal{E}'$;
    Calculate $w_{(v_{p'}, v_p)}$ based on Eq. (15);
    Assign $w_{(v_{p'}, v_p)}$ to edge $(v_{p'}, v_p)$ in $\mathcal{G}'$;
  end
  $\mathcal{G}' = (\mathcal{V}' \cup \{v_D, v_S\}, \mathcal{E}')$;
  Obtain minimum *s-t* cut $c^*_{\mathcal{G}'}$ of $\mathcal{G}'$ via the maximum flow method;
  $c^* = c^*_{\mathcal{G}'}$.
end

B. Performance Analysis

For linear neural networks, it is straightforward to see that the minimum *s-t* cut in the DAG is precisely equivalent to the optimal model partition in the AI model. Therefore, in this subsection, we focus on proving the equivalence between the optimal model partition in non-linear neural networks and the minimum *s-t* cut in the DAG $\mathcal{G}'$. To prove this equivalence, it begins with the following reasonable assumption.

Assumption 1: The server's computing capability is at least as strong as the device's. Specifically, for any layer, the computation latency on the server is no greater than that on the device, i.e.,

$$\xi_{D,v_i} - \xi_{S,v_i} \geq 0, \forall v_i \in \mathcal{V}. \tag{16}$$

Under Assumption 1, we can rigorously prove that the optimal model partition in the AI model is equivalent to finding the minimum *s-t* cut in the corresponding DAG. This relationship is formalized in the following theorem.

Theorem 1: The minimum *s-t* cut in the DAG $\mathcal{G}'$ corresponds exactly to the optimal model partition in the AI model.

Proof: Please refer to Appendix A, available online.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

VI. Block-Wise Model Partitioning Algorithm

In this section, we present a block-wise model partitioning algorithm to determine the optimal model partition for the AI models with block structures.

A. Intra-Block Cut Analysis

In an AI model, a block is a set of layers that is often reused multiple times. When an AI model contains block structures, the computational complexity of the optimal model partition can be further reduced. For example, the inception block is reused 9 times in GoogLeNet [27], the residual block is reused 8 and 16 times in ResNet18 and ResNet50 [28], respectively, and the dense block is reused 58 and 98 times in DenseNet121 and DenseNet201 [29], respectively. These blocks share the same internal architecture and transformation logic, although their input and output dimensions may vary depending on their position in the model. Because of this structural repetition, analyzing a block allows us to generalize its behavior across the other blocks.

1) Block Detection: To leverage block-level regularity, we first need to identify repeated blocks in the AI model. The detection process is based on the observation that many blocks exhibit a branching-aggregation structure, i.e., multiple parallel paths diverge from a single parent vertex and subsequently converge into a single vertex. When a parent vertex has multiple child vertices, these child vertices are added into a candidate set $\mathcal{V}_B^m$. We then continue tracking the successors of these child vertices until they all converge to the same vertex. At this stage, all intermediate successors and the final converged vertex are added into $\mathcal{V}_B^m$. The internal edges of this block are denoted as $\mathcal{E}_B^m$. Consequently, we obtain a block-level DAG denoted as $\mathcal{G}_B = (\mathcal{V}_B^m, \mathcal{E}_B^m)$. We continue this detection process towards the output end of the AI model, and if $\mathcal{G}_B$ appears multiple times, it is retained as a reusable unit. The detailed block-detection procedure is shown in Algorithm 3.

2) Intra-Block Cut Identification: The intra-block cut identification is designed to determine whether the optimal model partition intersects the internal edges of any block. The identification only relies on the sizes of smashed data within the block and does not require device or network parameters. If the optimal model partition intersects a block's internal edges, the block must be preserved in full. Otherwise, it can be safely abstracted as a single vertex of the DAG, simplifying the DAG structure and accelerating the cut selection process.

A block DAG is denoted by $\mathcal{G}_B = (\mathcal{V}_B, \mathcal{E}_B)$, where each vertex in $\mathcal{V}_B$ denotes a layer within the block and each directed edge in $\mathcal{E}_B$ indicates the flow of smashed data between layers. The input layer of the block is denoted as vertex v_{in}. As shown in Fig. 4, $\mathcal{V}_B = \{v_1, \ldots, v_7\}$ and v_{in} is the input layer. The vertices are unweighted, and each edge is weighted by the size of the smashed data from its parent vertex. The input and output layers of the block are treated as the source and sink of $\mathcal{G}_B$, respectively. Let c_B^{in} be the cut that separates the input layer from the block, and a_B^{in} be the size of smashed data that needs to be transmitted under this cut. Specifically, a_B^{in} is equal to the size of the smashed data generated at the input vertex v_{in}. Let $c_B^{\min}$ denote the minimum

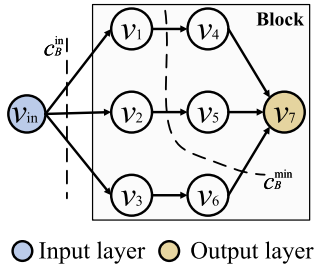


Fig. 4. An example of a block DAG, where each edge weight indicates the size of smashed data.

Algorithm 3: Block-Detection Procedure.

Input: $\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A)$;
Output: Set of blocks $\mathbf{V}_B = [\mathcal{V}_B^1, \ldots, \mathcal{V}_B^M]$;
Initialize $\mathbf{V}_B = \emptyset$, $M = 0$;
for *each* $v_i \in \mathcal{V}_A$ **do**
  if v_i *has multiple children* **then**
    $\mathcal{V}_B^m = \{$children of $v_i\}$;
    Set queue $\leftarrow$ children of v_i;
    while *successors of vertices in queue do not converge* **do**
      Add successors of queue into $\mathcal{V}_B^m$;
      Update queue $\leftarrow$ successors of queue;
    end
    Add the converged vertex into $\mathcal{V}_B^m$;
    Extract $\mathcal{E}_B^m$ as internal edges among $\mathcal{V}_B^m$;
  end
  $M \leftarrow M + 1$, $\mathbf{V}_B \leftarrow \mathbf{V}_B \cup \{\mathcal{V}_B^m\}$.
end

s-t cut of the block, and $a_B^{\min}$ be the size of smashed data that needs to be transmitted under this cut. $a_B^{\min}$ can be computed using standard maximum-flow algorithms [26], [30].

If $a_B^{\min} \geq a_B^{\text{in}}$, the optimal cut does not intersect any intra-block edges. Otherwise, it may intersect the intra-block edges. Under Assumption 1, we have the following theorem.

Theorem 2: If $a_B^{\min} \geq a_B^{\text{in}}$, the optimal cut does not intersect any intra-block edges.

Proof: Please refer to Appendix B, available online.

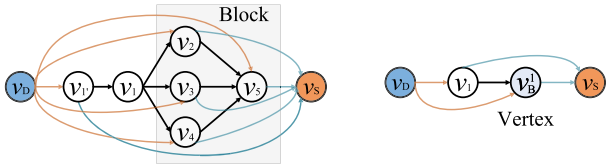
B. Block-Level Abstraction

When the optimal model partition does not intersect any intra-block edges, each repeated block in the AI model can be abstracted as a single vertex in the DAG. The detailed descriptions of the block-level abstraction are as follows.

1) Procedure: Firstly, let $\overline{\mathcal{G}} = (\overline{\mathcal{V}} \cup \{v_D, v_S\}, \overline{\mathcal{E}})$ denote a simplified DAG, where $\overline{\mathcal{V}} = \mathcal{V}$ and $\overline{\mathcal{E}} = \mathcal{E}$. Let M denote the number of blocks in $\overline{\mathcal{G}}$, and the set of blocks is indexed by $\mathbf{V}_B = [\mathcal{V}_B^1, \mathcal{V}_B^2, \ldots, \mathcal{V}_B^M]$. Each block is sequentially replaced by a single vertex in $\overline{\mathcal{G}}$. Let v_B^m denote the vertex that replaces block $\mathcal{V}_B^m$.

Secondly, the edge weights between v_B^m and other vertices in $\overline{\mathcal{G}}$ are calculated based on the original connections of block $\mathcal{V}_B^m$. The weight of the edge from v_B^m to sink vertex v_S, denoted

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.



(a) The DAG via DAG-based model partitioning algorithm

(b) The DAG via block-wise model partitioning algorithm

Fig. 5. An example of DAG transformation via general and block-wise model partitioning algorithms.

by $w_{(v_B^m,v_S)}$, is the sum of the device execution weights of all vertices within block $\mathcal{V}_B^m$, i.e.,

$$w_{(v_B^m,v_S)} = \sum_{v_b \in \mathcal{V}_B^m} w_{(v_b,v_S)}. \tag{17}$$

Thirdly, the weight of the edge from source vertex v_D to v_B^m in $\overline{\mathcal{G}}$, denoted by $w_{(v_D,v_B^m)}$, is the sum of the server execution weights of all vertices within this block, i.e.,

$$w_{(v_D,v_B^m)} = \sum_{v_b \in \mathcal{V}_B^m} w_{(v_D,v_b)}. \tag{18}$$

Fourthly, for any parent vertex of block $\mathcal{V}_B^m$, denoted as v_p, the weight of the edge from v_p to vertex v_B^m, denoted by $w_{(v_p,v_B^m)}$, is given by

$$w_{(v_p,v_B^m)} = w_{(v_p,v_b)}, \tag{19}$$

where v_b is any vertex in block $\mathcal{V}_B^m$.

Fifthly, for any child vertex of block $\mathcal{V}_B^m$, denoted as $v_{\tilde{b}}$, the weight of the edge from vertex v_B^m to vertex $v_{\tilde{b}}$, denoted by $w_{(v_B^m,v_{\tilde{b}})}$, is given by

$$w_{(v_B^m,v_{\tilde{b}})} = \sum_{v_b \in \mathcal{V}_B^m} w_{(v_b,v_{\tilde{b}})}. \tag{20}$$

The same abstraction procedure is applied to all blocks in the model.

2) Example: To illustrate the above transformation process more clearly, we provide an example as follows. Fig. 5(a) shows a DAG $\mathcal{G}'$ constructed by the general model partitioning algorithm, where a block is highlighted. Its corresponding DAG $\overline{\mathcal{G}}$ constructed by the block-wise model partitioning algorithm, is shown in Fig. 5(b). In $\overline{\mathcal{G}}$, the entire block is replaced by a single vertex v_B^1. The weight of the edge from this integrated vertex v_B^1 to the source vertex v_S, denoted as $w_{(v_B^1,v_S)}$, is the sum of the weights of all original edges from v_S to the vertices within the block:

$$w_{(v_B^1,v_S)} = w_{(v_2,v_S)} + w_{(v_3,v_S)} + w_{(v_4,v_S)} + w_{(v_5,v_S)}. \tag{21}$$

Similarly, the weight of the edge from the sink vertex v_D to v_B^1, denoted as $w_{(v_D,v_B^1)}$, is the sum of the weights of all original edges from the intra-block vertices to v_D:

$$w_{(v_D,v_B^1)} = w_{(v_D,v_2)} + w_{(v_D,v_3)} + w_{(v_D,v_4)} + w_{(v_D,v_5)}. \tag{22}$$

Algorithm 4: Block-Wise Model Partition Algorithm.

Input: $\mathcal{G} = (\mathcal{V} \cup \{v_D, v_S\}, \mathcal{E})$;
Output: c^*;
Calculate a_B^{in} and $a_B^{\min}$ based on the intra-block cut detection;
if $a_B^{\min} < a_B^{in}$ **then**
  // The optimal cut intersects intra-block edges.
  Obtain optimal cut c^* by inputting DAG $\mathcal{G}$ into Algorithm 2;
else
  // The optimal cut does not intersect any intra-block edges.
  $\overline{\mathcal{E}} \leftarrow \mathcal{E}$, $\overline{\mathcal{V}} \leftarrow \mathcal{V}$;
  Identify and define the block set $\mathbf{V}_B$;
  for $\mathcal{V}_B^m$ *in* $\mathbf{V}_B$ **do**
    Replace block $\mathcal{V}_B^m$ with vertex v_B^m in $\overline{\mathcal{V}}$;
    Add edges (v_n, v_B^m), (v_D, v_B^m), $(v_B^m, v_{\tilde{n}})$, and(v_B^m, v_S) into $\overline{\mathcal{E}}$;
    Calculate these edge weights connected to vertex v_B^m based on Eqs. (17)–(20);
    Assign edge weights to these edges;
  end
  $\overline{\mathcal{G}} = (\overline{\mathcal{V}} \cup \{v_D, v_S\}, \overline{\mathcal{E}})$;
  Obtain optimal cut c^* by inputting DAG $\overline{\mathcal{G}}$ into Algorithm 2.
end

The weight of the edge from vertex v_1 to integrated vertex v_B^1, denoted as $w_{(v_1,v_B^1)}$, is the weight of any one of the edges from v_1 to the intra-block vertices:

$$w_{(v_1,v_B^1)} = w_{(v_1,v_2)}. \tag{23}$$

For the DAG in Fig. 5(a), there are 17 edges and eight vertices. After the block integration by using the block-wise model partitioning algorithm, there are only five edges and four vertices in the simplified DAG, as shown in Fig. 5(b). The block-wise model partitioning algorithm can make the DAG more streamlined, thereby reducing the computational complexity of finding the optimal model partition.

C. Algorithm Execution

For an arbitrary AI model, we first run the block-detection procedure to check whether the model contains repeated blocks. If no repeated blocks are found, we directly apply the general model-partitioning algorithm. Otherwise, we perform intra-block cut identification to determine whether cutting within a block can be optimal (based on the condition derived in Section VI-A). If so, we keep the blocks expanded in the DAG. Otherwise, each repeated block is abstracted as a single vertex to simplify the DAG, and the optimal model partition can then be obtained more efficiently. The detailed procedure of the proposed block-wise model-partitioning algorithm is presented in Algorithm 4.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

D. Computational Complexity Analysis

The computational complexity of the proposed model-partitioning algorithms depends on the structure of the AI model, i.e., whether it is linear or non-linear. For a linear neural network, the optimal model partition can be directly obtained using a simple brute-force search method on the AI model, since no parent vertex has multiple outgoing edges. Therefore, the computational complexity of the proposed algorithm is $O(L)$ for a linear neural network, where L is the number of layers. For a non-linear neural network, we represent it as a DAG and compute the minimum *s-t* cut using a maximum flow algorithm. In this work, we adopt Dinic's algorithm [26], which is known for its efficiency in network flow problems. The computational complexity of Dinic's algorithm is $O(|\mathcal{V}|^2|\mathcal{E}|)$, where $|\mathcal{V}|$ is the number of vertices, and $|\mathcal{E}|$ is the number of edges in the DAG. Therefore, the computational complexity of our proposed algorithm is $O(|\mathcal{V}|^2|\mathcal{E}|)$ for non-linear neural networks.

If a brute-force search is applied to a non-linear neural network, the total number of candidate cut configurations is $2^{|\mathcal{V}|}$, where each subset denotes a possible cut. For each configuration, connectivity validation is performed, which incurs a complexity of $O(|\mathcal{V}| + |\mathcal{E}|)$. Hence, the overall computational complexity of the brute-force method is $O(2^{|\mathcal{V}|} \cdot (|\mathcal{V}| + |\mathcal{E}|))$, making it impractical for large-scale non-linear neural networks. In contrast, our proposed algorithm achieves significantly lower computational complexity compared with the brute-force search method when applied to non-linear neural networks.

E. Discussion

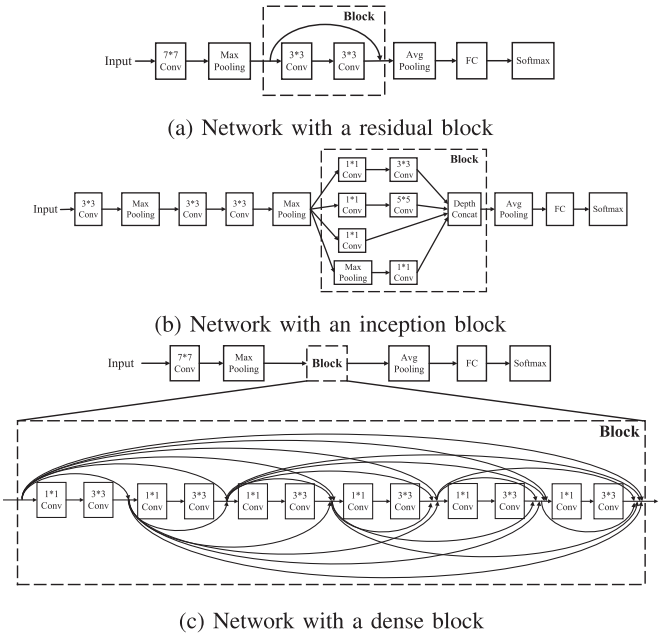
The proposed model partitioning algorithms can be extended to minimize the overall system energy consumption during model training or inference. This can be achieved by redefining the edge weights in the constructed DAG to represent the corresponding energy consumption. In this way, the minimum *s-t* cut naturally yields the energy-optimal model partition, enabling energy-aware split strategies for various training and inference scenarios. Furthermore, the proposed model partitioning algorithms can also be extended to transformer-based large language models (LLMs). In LLMs, components such as the embedding layer, transformer blocks, and classification head can be treated as distinct blocks. By applying the block-wise model partitioning algorithm, the optimal model partition of an LLM can be efficiently determined.

VII. Performance Evaluation

In this section, we first evaluate the performance of the proposed general and block-wise model partitioning algorithms for determining the optimal model partition. Then, we assess the performance of the proposed solution in wireless networks.

A. Performance of the Proposed Algorithms

1) Simulation Setup: In this subsection, the proposed general and block-wise model partitioning algorithms are evaluated by executing them on a server to obtain the optimal model partition.



(a) Network with a residual block

(b) Network with an inception block

(c) Network with a dense block

Fig. 6. The structures of block-based networks with residual, inception, and dense blocks, which exhibit different intra-block connections. (a)Network with a residual block. (b)Network with an inception block (c)Network with a dense block

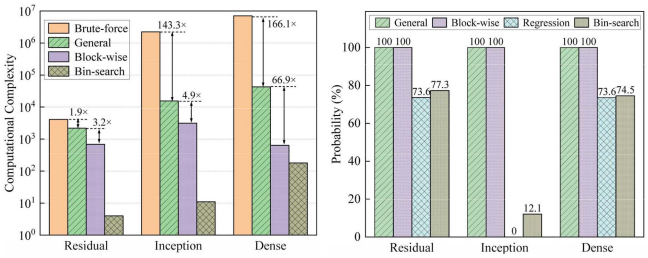
This server is equipped with an Intel Xeon(R) Gold 6226R CPU. The proposed partition algorithms are compared with the following benchmarks:

- **Brute-force** [10], which exhaustively examines all possible subsets to find the optimal model partition.
- **Regression** [21], which fits the quantitative relationship between the cut position and key factors in an AI model, thereby transforming the optimal model partitioning problem into a continuous optimization problem. The optimal model partition is then obtained by solving this optimization problem.
- **Bin-search** [13], which decomposes the DAG-based AI model into multiple parallel linear structures, and then applies binary search to determine the optimal partition for each linear path.

Since the regression method cannot handle non-linear neural networks directly, the block-level abstraction described in Section VI-B is applied to convert the model into a linear form, enabling compatibility with the regression approach.

To compare the cuts obtained by the proposed algorithms with those from the brute-force search method, we evaluate AI models containing non-linear blocks. However, due to the exponential computational complexity of brute-force search, applying it to non-linear models is impractical. Therefore, we consider three representative block-based networks, each incorporating a single type of non-linear block, as illustrated in Fig. 6. These blocks are: *residual blocks* [9], which introduce skip connections that add inputs directly to the outputs of convolutional layers and are widely used in ResNet; *inception blocks* [31], which combine multiple convolutional filters of varying kernel sizes and are commonly adopted in GoogLeNet; and *dense blocks* [32], which

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.



(a) Computational complexity (b) Probability of the optimal cut

Fig. 7. Performance comparison among the proposed algorithms and benchmarks on networks with a single block.

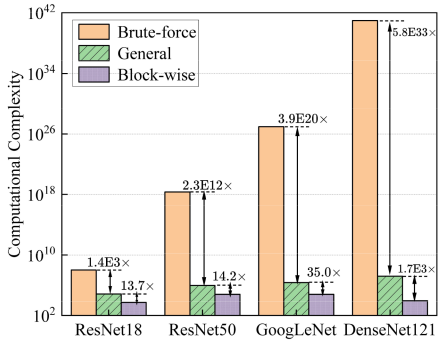


Fig. 8. Computational complexity comparison among the proposed algorithms and benchmarks on different AI models.

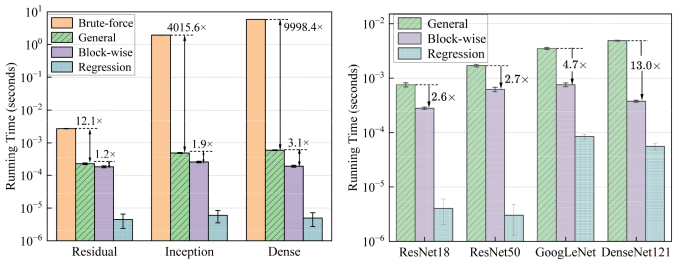
connect each layer to all subsequent layers to enhance feature reuse and are widely used in DenseNet.

In addition, to evaluate the of the proposed algorithms, experiments are conducted on four widely adopted AI models with varying depths and architectural patterns: *GoogLeNet*, a 22-layer model with nine inception blocks, an average layer output size of 22 MB, and a total computational load of 50 GFlops; *ResNet18*, an 18-layer model with eight residual blocks, an average layer output size of 1.1 MB, and a total computational load of 1.8 GFlops; *ResNet50*, a 50-layer model with 16 residual blocks, an average layer output size of 1.5 MB, and a total computational load of 3.7 GFlops; and *DenseNet121*, a 121-layer model with 58 dense blocks, an average layer output size of 29 MB, and a total computational load of 92 GFlops.

The *running time* is the time taken by the server to execute an algorithm and identify the optimal model partition for SL. The *probability of the optimal cut* refers to the probability that the cut obtained by an algorithm exactly matches the one obtained by the brute-force search method.

2) Effectiveness of the Proposed Algorithms: Since the regression method has a constant-time computational complexity of $\mathcal{O}(1)$, its theoretical complexity is omitted from the experimental analysis. Fig. 7(a) illustrates the computational complexity comparison between our proposed algorithms and baselines on networks with a single block. The general model partitioning algorithm significantly outperforms the brute-force search method, achieving reductions in computational complexity by $1.9\times$, $143.3\times$, and $166.1\times$ for networks with residual, inception, and dense blocks, respectively. Moreover, the block-wise model partitioning algorithm further streamlines the process, achieving additional reductions in complexity of $3.2\times$, $4.9\times$, and $66.9\times$ over the general model partitioning algorithm. This is because, in block-structured AI models, the block-wise model partitioning algorithm simplifies the DAG architecture, enabling more efficient identification of the optimal model partition.

Figure 7(b) presents the average probability of the optimal model partition over 1,000 simulation runs on networks with a single block. Although the regression and bin-search methods exhibit low computational complexity, they suffer from low probabilities of the optimal model partition. For example, on networks with a residual block, the probabilities of the regression and bin-search methods are only 73.6% and 77.3%, respectively. In particular, the probability of the regression method drops to zero on networks with an inception block. This is because both baselines fail to accurately capture the relationship between the partition point and model characteristics, particularly the size of the smashed data. In contrast, the proposed algorithm consistently identifies the optimal partition across all three network types. This is because the obtained cuts are provably optimal, as established in Theorem 1.



(a) Network with a block (b) AI model

Fig. 9. Running time comparison among the proposed algorithms and benchmarks.

Figure 8 presents the computational complexity of three algorithms on different AI models. The general model partitioning algorithm significantly reduces computational complexity compared to the brute-force search method. In contrast, the block-wise model partitioning algorithm further reduces it compared to the general model partitioning algorithm. Specifically, for DenseNet121, the general model partitioning algorithm achieves a 5.8×10^{33} reduction in computational complexity compared to the brute-force search method. The block-wise model partitioning algorithm further reduces the complexity by an additional factor of 1.7×10^3 compared to the general model partitioning algorithm. That is because DenseNet121 has the most repeated blocks among these models.

3) Running Time of the Proposed Algorithms: We evaluate the running time of the proposed algorithms on both networks, with single-block and full AI models. Due to the high complexity of full AI models, we only measure the running time of the brute-force search method on simpler block-based networks. All experimental results are averaged over 1,000 simulation runs to ensure statistical reliability.

Figure 9(a) compares the running time of the proposed algorithms against two baselines on the block-based networks.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

TABLE I
RUNNING TIME AND TRAINING DELAY COMPARISON

AI Model	Running Time (sec.)		Training Delay Per Iteration (sec.)
	General	Block-wise	
ResNet18	7.58×10^{-4}	2.81×10^{-4}	89.28
ResNet50	1.70×10^{-3}	6.22×10^{-4}	150.64
GoogLeNet	3.53×10^{-3}	7.58×10^{-4}	66.44
DenseNet121	4.91×10^{-3}	3.79×10^{-4}	78.90

Although these networks have relatively simple block structures, the brute-force search method still requires up to seconds of running time. Therefore, the brute-force search method is clearly impractical for practical AI models, since these models (e.g., ResNet18 and GoogLeNet) are significantly more complex than block-based networks. In contrast, the proposed algorithms significantly improve the computational efficiency of determining the optimal cut. The general model partitioning algorithm reduces the running time by $12.1\times$, $4{,}015.6\times$, and $9{,}998.4\times$ across the three networks, respectively, compared to the brute-force search method. Furthermore, the block-wise model partitioning algorithm achieves additional reductions of $1.2\times$, $1.9\times$, and $3.1\times$ compared to the general model partitioning algorithm. Both algorithms can determine the optimal cut for SL within milliseconds.

Figure 9(b) presents the running time of the proposed algorithms on four full AI models. The results show that the block-wise model partitioning algorithm consistently achieves lower running time than the general model partitioning algorithm across all AI models. Moreover, both algorithms can still determine the optimal cut within milliseconds, even for these non-linear AI models. This is well below the response latency tolerance of 200 ms [33], [34], indicating that the decision-making process can be completed in a timely manner even in dynamic environments. Although the regression method achieves the lowest overall running time, it fails to identify the optimal model partition, resulting in a significantly higher training delay. The limitations of the regression method will be presented in the following subsection.

To quantitatively compare the running time of the proposed algorithms with the model training delay per iteration, we conducted measurements on our hardware testbed. The experimental results are summarized in Table I. It can be observed that the average running time required to determine the cut is on the millisecond level, whereas the average training delay per iteration is on the minute level. Therefore, the running time is several orders of magnitude smaller than the training delay and can be safely ignored in practical training processes.

B. Proposed Scheme Performance in Dynamic Edge Networks

1) Simulation Setup: We evaluate the proposed algorithms using a custom-designed edge network simulator. The simulator's key components are listed below.

Network Topology: The proposed solution is tested in two representative edge network environments: mmWave and sub-6 GHz networks. Each edge network consists of a single base station equipped with a server and connected to 20 mobile devices via wireless links. For wireless data transmission, the n257 band is used in the mmWave network, and the n1 band is used in the sub-6 GHz network, following 3GPP specifications [35], [36].

Device Mobility: These devices, located within the base station's coverage area, request access and begin the device-side model training task at the start of each experimental trial. Each device continues to run the task while moving along a predefined trajectory at 30 *km/h*. The link bitrate is converted by the new radio channel quality indicator to the modulation and coding scheme mapping table [37]. Although the channel and link scheme does not account for complex propagation effects, it effectively simulates real-world mobile cellular dynamics, making it suitable for testing model partition. At each training round, the edge device closest to the base station is selected to participate in the task. To maintain fairness among devices, once a device is selected within an epoch, it cannot be selected again during the same epoch.

Channel Condition: Taking directional antenna gain into account, the transmit power is calculated as $P = P_e - 10\log_{10} N$, where P_e is the average effective isotropically radiated power (EIRP), and N is the number of beams. The server's average EIRP is fixed at 40 dBm for sub-6 GHz networks and 50 dBm for mmWave networks [38]. The number of beams is set to 16 for sub-6 GHz networks and 64 for mmWave networks. The large-scale path loss is defined as

$$PL(dB) = 32.5 + 20\log_{10}(f) + 10\eta\log_{10}(d) + \chi, \quad (24)$$

where f, η, d, and χ represent the carrier frequency, path-loss exponent, transmission distance, and shadow fading, respectively. The shadow fading follows $\mathcal{N}(0, \sigma^2)$. Furthermore, a small-scale fading is incorporated into the channel. Specifically, the effective path loss under Rayleigh fading is given by

$$PL_{\text{small}}(dB) = PL(dB) - 10\log_{10}(\psi), \quad (25)$$

where ψ denotes an independent random channel fading factor following an exponential distribution with unit mean [39]. In this paper, the channel condition is set to three states: *Good*, *Normal*, and *Poor*, corresponding to σ values of 2 dB, 4 dB, and 6 dB, respectively.

In practical wireless systems, the base station continuously collects channel quality information (e.g., channel quality indicators (CQI) and buffer status reports (BSR)) as part of standard radio resource management procedures. Our method leverages these existing measurements to estimate the uplink and downlink transmission rates, without requiring any additional signaling mechanisms. According to measurements in the 5G radio access network (RAN), the average packet round-trip time (RTT), which reflects the latency of channel estimation and link adaptation, is approximately 9.23 ms [34]. This indicates that the communication overhead for collecting network state information is relatively small.

Computation Delay Profiles: To accurately measure computation delay, develop a standalone Python-based latency profiling tool for *PyTorch*. This tool traverses the AI model and

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

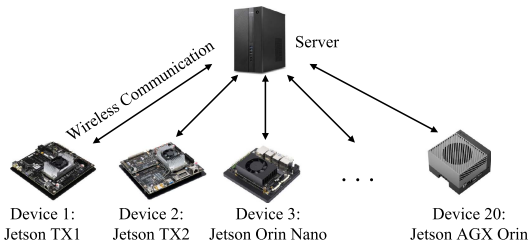


Fig. 10. Prototype system of SL.

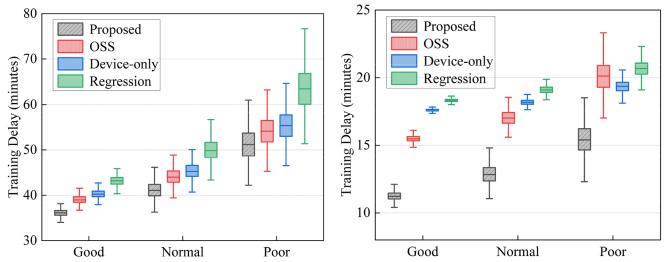


Fig. 11. Training delay per epoch of the proposed solution and benchmarks under large-scale path loss. (a)Sub-6 GHz network. (b)mmWave network

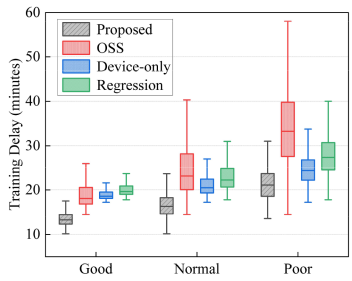


Fig. 12. Training delay per epoch of the proposed solution and benchmarks in the mmWave network under a Rayleigh fading channel.

registers forward and backward hooks on each layer, following the method introduced in [6]. These hooks record per-layer processing times during training, enabling fine-grained profiling of training delays. For communication delay estimation, we use PyTorch's built-in `torchstat` module to record each layer's parameter size and output size. All training and profiling experiments are performed on graphics processing units (GPUs).

Our prototype system is illustrated in Fig. 10, consisting of a server and 20 devices configured as follows: (*i*) a personal computer (PC) with 1× Nvidia RTX A6000 GPU; (*ii*) five Jetson TX1, each with 256-core Nvidia Maxwell GPU; (*iii*) five Jetson TX2, each with 256-core Nvidia Pascal GPU; (*iv*) five Jetson Orin Nano, each with 1024-core Nvidia Ampere architecture GPU; and (*iv*) five Jetson AGX Orin, each with 2048-core Nvidia Ampere architecture GPU. Through the prototype system, we measure training delay profiles on both the server and heterogeneous mobile devices, demonstrating the system's ability to adapt to real-world variations in device capabilities. The performance of the proposed solution is compared with three benchmarks:

- **Optimal static partition (OSS)** [18], which refers to the static cut that minimizes training latency among all possible fixed cuts.
- **Device-only**, where the entire AI model is trained on the device, while the server is only responsible for data transmission.
- **Regression** [21] and **Bin-search** [13], as mentioned in the previous subsection.

2) Impact of Channel Conditions: To evaluate the performance of the proposed solution under dynamic wireless environments, we train GoogLeNet over 300 simulation runs across three channel conditions.

Figure 11 illustrates the model training delay per epoch for each method under both sub-6 GHz and mmWave networks. The regression method performs poorly due to its suboptimal model partition. In contrast, the proposed solution consistently achieves the lowest training delay across all channel conditions. In the sub-6 GHz network, the proposed solution reduces training delay by 11.40%, 12.38%, and 19.34% compared to the OSS, device-only, and regression methods, respectively. In the mmWave network, the proposed solution also outperforms all benchmarks, achieving reductions in training delay of 27.38%, 36.09%, and 38.61%, respectively. These results demonstrate that the proposed solution can rapidly determine the optimal model partition under real-time network conditions, particularly in highly dynamic environments.

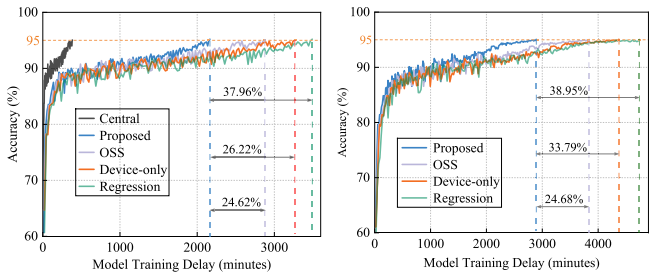
Figure 12 shows the performance under Rayleigh fading channels in the mmWave network. As we can see, the training delay per epoch of the OSS method is significantly affected by Rayleigh fading, exhibiting large fluctuations. In contrast, the proposed solution remains relatively stable under the same conditions. This is because the transmission rate under Rayleigh fading varies much more dramatically than that under large-scale path loss. Since the OSS method uses a fixed cut, it cannot adapt to rapid channel variations, leading to substantial fluctuations in transmission delay. In contrast, the proposed solution dynamically adjusts the cut according to channel conditions, thereby maintaining a more stable training delay per epoch even in the presence of Rayleigh fading. Furthermore, under Rayleigh fading channels, the proposed solution consistently outperforms all baselines across different channel conditions.

3) Impact of Data Distribution: To evaluate the performance of the proposed solution under different data distributions, we train GoogLeNet and compare it with four benchmarks: *central*, which runs the entire model on the server, *OSS*, *device-only*, and *regression*. The experiments are conducted in the mmWave network with normal channel conditions. Two types of data distributions are considered, i.e., independent and identically distributed (IID) and non-IID. In the IID setting, each device receives an equal number of samples from each class in the dataset. In this experiment, training delay is defined as the time required for the model to reach 95% accuracy under each algorithm. As shown in Fig. 13(a), the proposed solution reduces the training delay by 37.96%, 26.22%, and 24.62% compared to the regression, device-only, and OSS benchmarks, respectively.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

TABLE II
TRAINING DELAY COMPARISON AMONG DIFFERENT SCHEMES

Model	Method	Training Delay (minutes)			
		CIFAR-10		CIFAR-100	
		IID (Accuracy 95%)	non-IID (Accuracy 95%)	IID (Accuracy 79 ± 1%)	non-IID (Accuracy 78 ± 1%)
GoogLeNet	OSS	2,872.04 (**1.33**×)	3,839.34 (**1.33**×)	3,201.44 (**1.29**×)	3,601.12 (**1.35**×)
	Device-only	3,265.28 (**1.51**×)	4,367.28 (**1.51**×)	3,851.60 (**1.55**×)	4,105.14 (**1.54**×)
	Regression	3,489.76 (**1.61**×)	4,737.32 (**1.64**×)	3,911.03 (**1.57**×)	4,390.73 (**1.65**×)
	Bin-search	2,827.37 (**1.31**×)	3,754.24 (**1.30**×)	3,141.59 (**1.26**×)	3,487.14 (**1.31**×)
	Proposed	2,164.36	2,891.81	2,489.61	2,660.29
ResNet18	OSS	2,905.48 (**1.28**×)	3,272.18 (**1.29**×)	3,127.26 (**1.25**×)	3,327.46 (**1.34**×)
	Device-only	3,331.11 (**1.47**×)	3,640.04 (**1.43**×)	3,407.30 (**1.36**×)	3,361.40 (**1.35**×)
	Regression	3,201.30 (**1.41**×)	3,502.76 (**1.38**×)	3,440.95 (**1.38**×)	3,318.70 (**1.34**×)
	Bin-search	2,894.61 (**1.28**×)	3,169.16 (**1.25**×)	3,016.76 (**1.21**×)	3,286.74 (**1.32**×)
	Proposed	2,265.09	2,540.26	2,499.24	2,484.79
ResNet50	OSS	3,242.20 (**1.33**×)	4,208.29 (**1.16**×)	2,790.26 (**1.28**×)	3,842.83 (**1.36**×)
	Device-only	3,685.03 (**1.51**×)	5,586.54 (**1.56**×)	3,187.27 (**1.46**×)	4,057.17 (**1.44**×)
	Regression	3,639.25 (**1.49**×)	5,490.38 (**1.52**×)	3,113.14 (**1.43**×)	4,038.32 (**1.43**×)
	Bin-search	3,324.41 (**1.36**×)	4,346.26 (**1.20**×)	2,904.23 (**1.33**×)	3,994.31 (**1.42**×)
	Proposed	2,445.16	3,611.98	2,179.92	2,819.09
DenseNet121	OSS	3,717.68 (**1.16**×)	4,215.98 (**1.15**×)	3,002.23 (**1.28**×)	4,300.22 (**1.33**×)
	Device-only	4,940.63 (**1.55**×)	5,744.91 (**1.57**×)	3,420.54 (**1.46**×)	5,021.96 (**1.55**×)
	Regression	4,921.35 (**1.54**×)	5,702.53 (**1.56**×)	3,374.06 (**1.44**×)	4,954.29 (**1.53**×)
	Proposed	3,196.94	3,652.77	2,341.60	3,232.60



(a) IID (b) non-IID

Fig. 13. Overall training delay comparison among the proposed solution and benchmarks when training GoogLeNet. (a)IID. (b)non-IID

In the non-IID dataset, we employ the method described in [40] to synthesize data for different devices. Specifically, each device samples data from a subset of categories, where the category proportions $\mathcal{Q} = q_1, q_2, \ldots, q_M$ follow a Dirichlet distribution, i.e., $\mathcal{Q} \sim Dir(\gamma p)$. Here, p denotes the label distribution, and γ controls the degree of data heterogeneity across devices. In this experiment, we set $\gamma = 0.5$. As shown in Fig. 13(b), the proposed solution reduces the training delay by 38.95%, 33.79%, and 24.68% compared with the regression, device-only, and OSS benchmarks, respectively. In the device-only method, although only a single transmission is required after completing the full local iteration to upload the model parameters to the server, the overall training delay remains relatively high. This is because mobile devices typically have limited computational capability, leading to longer local training delay compared to the proposed solution. Moreover, the size of the transmitted model parameters is larger than that of the smashed data at the optimal cut determined by the proposed solution. As a result, both computation and communication delays are increased in the device-only approach. This demonstrates that under non-IID data conditions, the proposed solution offers greater advantages, highlighting its robustness to heterogeneous data distributions.

4) Impact of AI Models: All models are trained over the mmWave network with normal channel conditions. To ensure a fair comparison, all schemes are required to achieve their predefined accuracy thresholds using the same datasets, data distributions, and AI models. These thresholds may vary slightly across models due to inherent differences in their performance. For example, on the CIFAR-100 dataset with a non-IID distribution and ResNet18, the accuracy threshold is set to 77%, whereas with ResNet50 under the same conditions, the threshold is 78%.

The overall training delay for each method to reach its accuracy threshold is summarized in Table II. Due to the extremely high computational complexity of the bin-search method when determining the optimal partition for DenseNet121, its training delay is not reported in the table. The proposed solution substantially reduces training delay compared with benchmarks across all AI models, datasets, and data distributions. For example, when training ResNet50 on the CIFAR-100 dataset with a non-IID data distribution, the proposed solution reduces the overall training delays by 1.36×, 1.44×, 1.43×, and 1.42× compared with the OSS, device-only, regression, and bin-search methods, respectively. In addition, we further evaluate the performance on GPT-2 under the mmWave network with normal channel conditions. GPT-2 is trained on the CARER dataset [41] with a non-IID data distribution. As illustrated in Fig. 14, the proposed solution reduces the overall training delay by 8.62%, 23.48%, and 73.42% compared with the OSS, regression, and device-only methods, respectively. These performance gains are primarily due to the proposed solution's ability to select the optimal model partition that minimizes overall training delay, thereby mitigating resource bottlenecks and accelerating the training process.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

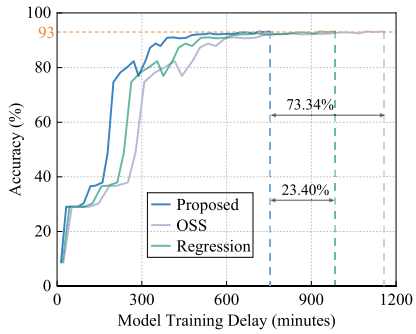
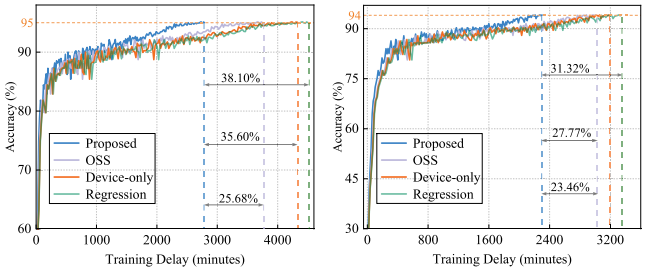


Fig. 14. Overall training delay comparison among the proposed solution and benchmarks when training GPT-2 on the CARER dataset.



(a) Training delay (10 devices) (b) Training delay (40 devices)

Fig. 15. Overall training delay comparison under different numbers of devices when training GoogLeNet with a non-IID CIFAR-10 dataset in the mmWave network. (a)Training delay (10 devices). (b)Training delay (40 devices)

5) Impact of Network Sizes: To evaluate the robustness of the proposed approach against network sizes, the training delay under different numbers of devices is measured when training GoogLeNet with a non-IID CIFAR-10 dataset in the mmWave network. As shown in Fig. 15, when the number of devices is 10 or 40, the proposed scheme consistently outperforms the baseline methods, achieving lower training delay. Specifically, the proposed solution reduces the training delay by at least 25.68% and 23.46% compared with the baselines when the number of devices is 10 and 40, respectively. Despite the increase in the number of devices, the proposed approach maintains fast convergence, demonstrating its robustness against network sizes.

6) Communication Overhead: To clarify the communication overhead introduced by model splitting, we compare the device-side computing delay, server-side computing delay, and communication delay under different methods. Specifically, the results are obtained when training GoogLeNet over the mmWave network with a batch size of 32, where the reported delays correspond to the total computation and communication delay for completing two iterations jointly executed by the device and the server. As shown in Fig. 16, the proposed solution reduces the communication delay by 23.82%, 30.83%, and 45.96% compared with the bin-search, regression, and OSS methods, respectively. This improvement is attributed to the optimal cut selected by the proposed solution, which effectively reduces the size of the intermediate data transmitted between the device and the server. In addition, although the device-only method incurs lower communication delay than the proposed scheme, it suffers from significantly higher device-side computation delay. As a result, its overall training delay is still larger than that of the proposed scheme.

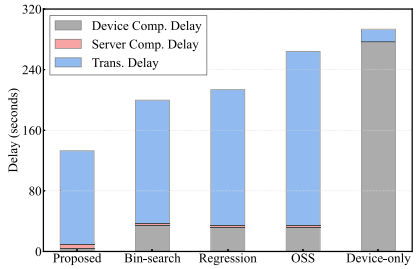


Fig. 16. Computing delay and transmission delay comparison.

VIII. Conclusion

In this paper, we have investigated the optimal model partitioning problem to minimize the training delay of SL in edge networks. We have represented an arbitrary AI model as a DAG and proposed a general model partitioning algorithm to determine the optimal model partition. For AI models with block structures, we have proposed a low-complexity block-wise partitioning algorithm to efficiently determine the optimal model partition. Extensive experimental results have validated that the proposed scheme can find the optimal model partition within the millisecond. Due to fast and accurate identification of the optimal model partition, the proposed solution can be applied to facilitate AI model training in edge networks with a large number of resource-constrained mobile devices. For future work, we will investigate the multi-partition problem of SL in edge networks.

References

[1] S. Wang, Y. Ruan, Y. Tu, S. Wagle, C. G. Brinton, and C. Joe-Wong, "Network-aware optimization of distributed learning for fog computing," *IEEE/ACM Trans. Netw.*, vol. 29, no. 5, pp. 2019–2032, Oct. 2021.

[2] K. B. Letaief, Y. Shi, J. Lu, and J. Lu, "Edge artificial intelligence for 6G: Vision, enabling technologies, and applications," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 5–36, Jan. 2022.

[3] X. Shen, J. Gao, W. Wu, M. Li, C. Zhou, and W. Zhuang, "Holistic network virtualization and pervasive network intelligence for 6G," *IEEE Commun. Surveys Tut.*, vol. 24, no. 1, pp. 1–30, Firstquarter 2022.

[4] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *J. Netw. Comput. Appl.*, vol. 116, pp. 1–8, Aug. 2018.

[5] J. Sun et al., "An efficient privacy-aware split learning framework for satellite communications," *IEEE J. Sel. Areas Commun.*, vol. 42, no. 12, pp. 3355–3365, Dec. 2024.

[6] S. Wang, X. Zhang, H. Uchiyama, and H. Matsuda, "HiveMind: Towards cellular native machine learning model splitting," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 2, pp. 626–640, Feb. 2022.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. NeurIPS*, 2012, pp. 1–9.

[9] S. Targ, D. Almeida, and K. Lyman, "Resnet in Resnet: Generalizing residual architectures," 2016, *arXiv:1603.08029*.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.

[10] Z. Lin et al., "Efficient parallel split learning over resource-constrained wireless edge networks," *IEEE Trans. Mobile Comput.*, vol. 23, no. 10, pp. 9224–9239, Oct. 2024.
[11] J.-A. Lim, J. Lee, J. Kwak, and Y. Kim, "Cutting-edge inference: Dynamic DNN model partitioning and resource scaling for mobile AI," *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 3300–3316, Nov./Dec. 2024.
[12] J. Li, W. Liang, Y. Li, Z. Xu, X. Jia, and S. Guo, "Throughput maximization of delay-aware DNN inference in edge computing by exploring DNN model partitioning and inference parallelism," *IEEE Trans. Mobile Comput.*, vol. 22, no. 5, pp. 3017–3030, May 2023.
[13] Y. Duan and J. Wu, "Joint optimization of DNN partition and scheduling for mobile cloud computing," in *Proc. ACM ICPP*, New York, NY, USA, Oct. 2021, pp. 1–10.
[14] 3rd Generation Partnership Project (3GPP), "5G system (5GS); study on traffic characteristics and performance requirements for AI/ML model transfer," 3GPP, Sophia Antipolis, France, Tech. Rep. TR 22.874 V18.2.0, 2021.
[15] V. Ninkovic, D. Vukobratovic, D. Miskovic, and M. Zennaro, "COMSPLIT: A communication–aware split learning design for heterogeneous IoT platforms," *IEEE Internet Things J.*, vol. 12, no. 5, pp. 5305–5319, Mar. 2024.
[16] J. Yan, S. Bi, and Y.-J. A. Zhang, "Optimal model placement and online model splitting for device-edge co-inference," *IEEE Wireless Commun.*, vol. 21, no. 10, pp. 8354–8367, Oct. 2022.
[17] W. Wu et al., "Split learning over wireless networks: Parallel design and resource management," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 1051–1066, Apr. 2023.
[18] C. Xu, J. Li, Y. Liu, Y. Ling, and M. Wen, "Accelerating split federated learning over wireless communication networks," *IEEE Trans. Wireless Commun.*, vol. 23, no. 6, pp. 5587–5599, Jun. 2024.
[19] J. Dachille, C. Huang, and X. Liu, "The impact of cut layer selection in split federated learning," 2024, *arXiv:2412.15536*.
[20] J. Lee, J. Cho, W. Lee, M. Seif, and H. V. Poor, "Game-theoretic joint incentive and cut layer selection mechanism in split federated learning," 2024, *arXiv:2412.07813*.
[21] Y. Wen, G. Zhang, K. Wang, and K. Yang, "Training latency minimization for model-splitting allowed federated edge learning," *IEEE Trans. Netw. Sci. Eng.*, vol. 12, no. 3, pp. 2081–2092, May/Jun. 2025.
[22] H. Liang et al., "DNN surgery: Accelerating DNN inference on the edge through layer partitioning," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 3111–3125, Jul./Sep. 2023.
[23] Y. Dai et al., "A survey of graph-based resource management in wireless networks - Part II: Learning approaches," *IEEE Trans. Cogn. Commun. Netw.*, vol. 11, no. 4, pp. 2101–2122, Aug. 2025.
[24] L. Wu, G. Gao, J. Yu, F. Zhou, Y. Yang, and T. Wang, "PDD: Partitioning DAG-topology DNNs for streaming tasks," *IEEE Internet Things J.*, vol. 11, no. 6, pp. 9258–9268, Mar. 2024.
[25] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, Apr. 2017.
[26] Y. Dinitz, "Dinitz' algorithm: The Original Version And Even's Version," Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 218–240, doi:10.1007/11685654_10.
[27] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Boston, Massachusetts, USA, Jun. 2015, pp. 1–9.
[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Las Vegas, Nevada, USA, Jun. 2016, pp. 770–778.
[29] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Honolulu, Hawaii, USA, Jul. 2017, pp. 4700–4708.
[30] M. Stoer and F. Wagner, "A simple min cut algorithm," in *Proc. Eur. Symp. Algorithms*, Sep. 1994, pp. 141–147.
[31] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *Proc. Conf. Assoc. Advance. Artif. Intell.*, Palo Alto, USA, Feb. 2017, pp. 4278–4284.
[32] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger, "CondenseNet: An efficient DenseNet using learned group convolutions," in *Proc.IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Salt Lake City, UT, USA, Jun. 2018, pp. 2752–2761.
[33] Z. Tan, Y. Li, Q. Li, Z. Zhang, Z. Li, and S. Lu, "Supporting mobile VR in LTE networks: How close are we," in *Proc. ACM Meas. Anal. Comput.*, vol. 2, no. 1, pp. 1–31, Mar. 2018.
[34] D. Xu, R. Lin, H. Zhang, A. Zhou, and H. Ma, "Bridging cross-layer interactions between 5G RAN and MEC for latency-critical video analytics," *IEEE/ACM Trans. Netw.*, vol. 33, no. 4, pp. 1614–1629, Aug. 2025.
[35] 3rd Generation Partnership Project (3GPP), "NR; user equipment (UE) radio transmission and reception," 3GPP, Sophia Antipolis, France, Tech. Specification 38.101-1 V18.6.0, 2024.
[36] M. L. Hakim, M. T. Islam, and T. Alam, "Incident angle stable broadband conformal mm-Wave FSS for 5G (n257, n258, n260, and n261) band EMI shielding application," *IEEE Antennas Wirel. Propag. Lett.*, vol. 23, no. 2, pp. 488–492, Feb. 2024.
[37] 3rd Generation Partnership Project (3GPP), "NR; physical layer procedures for data," *3GPP, Sophia Antipolis, France,Tech. Specification 38.214 V17.3.0*, 2022.
[38] Y. Wang, C. Chen, and X. Chu, "Performance analysis for hybrid Sub-6GHz-mmWave-THz networks with downlink and uplink decoupled cell association," *IEEE Trans. Wireless Commun.*, vol. 24, no. 8, pp. 6658–6673, Aug. 2025.
[39] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, Nov. 2020.
[40] C. Feng, H. H. Yang, D. Hu, Z. Zhao, T. Q. S. Quek, and G. Min, "Mobility-aware cluster federated learning in hierarchical wireless networks," *IEEE Trans. Wireless Commun.*, vol. 21, no. 10, pp. 8441–8458, Oct. 2022.
[41] E. Saravia, H.-C. T. Liu, Y.-H. Huang, J. Wu, and Y.-S. Chen, "CARER: Contextualized affect representations for emotion recognition," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, Brussels, Belgium, 2018, pp. 3687–3697.

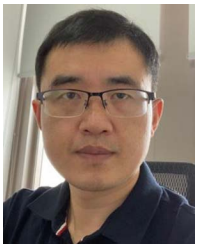


Zuguang Li (Graduate Student Member) received the BS and MS degrees in electronic information science and technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2020 and 2023, respectively. He is currently working toward the PhD degree in information and communication engineering with Harbin Institute of Technology, Shenzhen, China. His research interests include 6G networks, network intelligence, and blockchain technologies.



Wen Wu (Senior Member, IEEE) received the PhD degree in electrical and computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 2019. He is currently an associate researcher with the Frontier Research Center, Pengcheng Laboratory, Shenzhen, China. He has authored or coauthored 140 refereed IEEE journal and conference papers, as well as 6 books/book chapters, filed 10+ PCT patents, and issued 2 US patents. His research interests include 6G networks, edge LLM, split learning, network intelligence, and network virtualization. He was the recipient World Top 2% Scientist Award in 2023-2025, OJVT Best Paper Award from the IEEE Vehicular Technology Society, and the Award for Excellence (Early Career Researcher) from the IEEE Hyper-Intelligence Technical Committee, ACM Mobicom Best Workshop Paper Award, USENIX Security Distinguished Paper Award, ComSoc TEA Excellence Camp Runner-Up Award, IEEE ICCT Best Student Paper Award, and IEEE CIC/ICCC Best Paper Award. He was as Track Co-Chair for IEEE VTC 2022-2025, and Workshop Co-Chair for IEEE INFOCOM 2022-2025, IEEE Globecom 2024, IEEE MASS 2025, and IEEE ICCC 2023-2025. He is an Editor of *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Cognitive Communications and Networking*, *Journal of Communications and Information Networks*, and Springer PPNA. He was a Guest Editor of *IEEE Networking Letters*, HINDAWI WCMC, *China Communications*, and *Electronics*.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.



Shaohua Wu (Senior Member, IEEE) received the PhD degree in communication engineering from Harbin Institute of Technology in 2009. From 2009 to 2011, he held a postdoctoral position with the Department of Electronics and Information Engineering, Shenzhen Graduate School, Harbin Institute of Technology, where he has been with since 2012. From 2014 to 2015, he was a visiting researcher with BBCR, University of Waterloo, ON, Canada. He is currently a full professor with the Harbin Institute of Technology (Shenzhen), China. He is also a professor with Pengcheng Laboratory, Shenzhen, China. He has authored or coauthored over 200 papers in these fields and holds over 50 Chinese patents. His research interests include satellite and space communications, advanced channel coding techniques, space-air-ground-sea integrated networks, and B5G/6G wireless transmission technologies.



Xuemin (Sherman) Shen (Fellow, IEEE) received the PhD degree in electrical engineering from Rutgers University, New Brunswick, NJ, USA, in 1990. He is currently a university professor with the Department of Electrical and Computer Engineering, University of Waterloo, ON, Canada. His research interests include network resource management, wireless network security, Internet of Things, 5G and beyond, and vehicular networks. He is a registered professional engineer of Ontario, Canada, an engineering Institute of Canada Fellow, Canadian Academy of Engineering fellow, Royal Society of Canada fellow, a Chinese Academy of Engineering Foreign member, and an Engineering Academy of Japan International Fellow. He was the recipient of the "West Lake Friendship Award" from Zhejiang Province in 2023, the Canadian Award for Telecommunications Research from the Canadian Society of Information Theory (CSIT) in 2021, the R.A. Fessenden Award in 2019 from IEEE, Canada, Award of Merit from the Federation of Chinese Canadian Professionals (Ontario) in 2019, James Evans Avant Garde Award in 2018 from the IEEE Vehicular Technology Society, Joseph LoCicero Award in 2015 and Education Award in 2017 from the IEEE Communications Society (ComSoc), and Technical Recognition Award from Wireless Communications Technical Committee (2019) and AHSN Technical Committee (2013). He was also the recipient the Excellent Graduate Supervision Award in 2006 from the University of Waterloo and the Premier's Research Excellence Award (PREA) in 2003 from the Province of Ontario, Canada. He is/was as the general chair for the 6G Global Conference'23, and ACM Mobihoc'15, Technical Program Ccommittee chair/co-chair for IEEE Globecom'24, '16 and '07, IEEE Infocom'14, and IEEE VTC'10 Fall. He is the Past President of the IEEE ComSoc, the Vice President for Technical & Educational Activities, Vice President for Publications, member-at-Large on the Board of Governors, Chair of the Distinguished Lecturer Selection Committee, and member of IEEE Fellow Selection Committee of the ComSoc. He was a the Editor-in-Chief of the *IEEE Internet of Things Journal*, *IEEE Network*, and *IET Communications*.

Authorized licensed use limited to: Nanjing University. Downloaded on October 30,2026 at 06:13:18 UTC from IEEE Xplore. Restrictions apply.