applied sciences

MDPI

Article

Distributed Inference Models and Algorithms for Heterogeneous Edge Systems Using Deep Learning

Qingqing Yuan and Zhihua Li *

School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, China; londey@163.com
* Correspondence: zhli@jiangnan.edu.cn

Abstract: Computations performed by using convolutional layers in deep learning require significant resources; thus, their scope of applicability is limited. When deep neural network models are employed in an edge-computing system, the limited computational power and storage resources of edge devices can degrade inference performance, require a considerable amount of computation time, and result in increased energy consumption. To address these issues, this study presents a convolutional-layer partitioning model, based on the fused tile partitioning (FTP) algorithm, for enhancing the distributed inference capabilities of edge devices. First, a resource-adaptive workload-partitioning optimization model is designed to promote load balancing across heterogeneous edge systems. Next, the FTP algorithm is improved, leading to a new layer-fused partitioning method that is used to solve the optimization model. The results of simulation experiments show that the proposed convolutional-layer partitioning method effectively improves the inference performance of edge devices. When five edge devices are used, the speed of the proposed method becomes 1.65–3.48 times those of existing algorithms.

Keywords: edge computing; distributed inference; edge intelligence; resource allocation

Academic Editors: Hyokyung Bahn, Jože Guna and Raul Parada

Received: 10 November 2024
Revised: 8 January 2025
Accepted: 21 January 2025
Published: 22 January 2025

Citation: Yuan, Q.; Li, Z. Distributed Inference Models and Algorithms for Heterogeneous Edge Systems Using Deep Learning. *Appl. Sci.* **2025**, *15*, 1097. https://doi.org/10.3390/app15031097

1. Introduction

Deep neural networks (DNNs) are vital in computationally intensive and memory-intensive applications, such as computer vision, natural language processing, and speech recognition [1]. However, the computational complexity of DNNs is increasing because of the increasing demand for higher precision in these applications. This increasing computational complexity requires significant computational resources, which inevitably limit the applications of DNNs. This problem is conventionally mitigated by offloading computationally intensive tasks to cloud data centers, with edge devices receiving the processed results. However, this approach introduces network latency and network delays. Owing to the rapid development of artificial intelligence and the Internet of Things, edge computing has emerged as an important strategy for processing large-scale data. In edge computing, edge devices process data in real time at the source, thereby reducing latency and alleviating the burden on cloud data centers. Achieving efficient distributed inference on resource-constrained edge devices has become an important challenge in current research.

In recent years, numerous studies [2–17] have focused on developing various inference methods for edge devices. These methods aim to alleviate the conflict between the resource requirements of DNN model deployment and the limited computational resources of edge devices. These methods can be categorized into three types: methods based on DNN model modification, methods of cloud–edge collaboration, and methods based on distributed inference. Certain studies [2–8] were primarily focused on modifying the

DNN architecture. Model compression algorithms [3–6] have been employed to eliminate unnecessary parameters for resource-constrained edge devices. Knowledge distillation methods have also been used to appropriately reduce model complexity [7,8]. However, these methods [3–8] are only applicable to homogeneous computing systems. When multiple types of edge devices are involved, the models must be modified separately for each type; thus, these approaches are ineffective in heterogeneous edge computing. Studies [9–11] have investigated cloud-edge collaboration methods to partition DNN models, offloading computationally intensive components to cloud data centers to alleviate the workload on edge devices. However, these approaches are constrained by the network bandwidth, leading to high latency and unreliable performance. Furthermore, they do not account for network fluctuations and, thus, are unsuitable for real-time inference in heterogeneous edge-computing systems. The distributed-inference-based approach is adopted to address these limitations and improve the real-time adaptability of DNN inference on edge devices [12–17]. Experimental results show that this method leverages the collaboration between edge devices to enhance the overall inference performance.

However, in [14,15,17], only applications involving homogeneous edge devices were considered; thus, the scalability of edge device inference was limited. In [12,13,16], complex DNN models for heterogeneous edge computing were not sufficiently explored; this can lead to high latency and energy consumption on edge devices. By contrast, in this study, the workload-partitioning problem for heterogeneous edge systems is addressed. This is achieved by comprehensively considering the computational capabilities and the processing and transmission powers of these devices. To achieve the optimization objective of reducing inference latency and energy consumption, the workload-partitioning problem is formulated by applying a resource-adaptive workload-partitioning optimization model. The fused tile partitioning (FTP) algorithm [14] exhibits good adaptability when partitioning inference tasks for different DNN models. It also significantly improves inference performance on edge devices, so an enhanced version of this algorithm is proposed to better suit heterogeneous edge-computing systems. A new layer-fused partitioning (LFP) method is designed based on the enhanced FTP algorithm used to solve the optimization problem. LFP does not require any structural modifications or tuning of the given DNN model, nor does it compromise model accuracy, as it preserves the input data and model parameters of the given DNN model.

The main contributions of this study are summarized as follows: (1) a resource-adaptive workload-partitioning optimization model for deploying DNNs in resource-constrained heterogeneous edge systems is established to address load imbalances across devices; (2) a utility function is formulated to optimize latency and energy consumption. The workload-partitioning problem in heterogeneous edge-computing systems is then expressed as an integer linear programming (ILP) problem; amd (3) an LFP method is designed as an enhanced FTP-based method that yields near-optimal solutions to the workload-partitioning problem and significantly enhances the inference performance on edge devices. Simulation results show that the LFP method offers advantages over existent algorithms.

2. Related Work

Numerous techniques and theoretical models have been proposed to address the problem of accelerating DNN inference on resource-constrained edge devices [2–17]. Based on the relationship between edge devices and DNN models, these techniques can be categorized into three types: methods based on DNN model modification, methods of cloud–edge collaboration, and methods based on distributed inference.

2.1. DNN Model Modification

Various model compression techniques have been proposed to facilitate DNN inference directly on edge devices [5–8,18,19]. For example, weight-pruning methods [5,6] are used to eliminate redundant and insignificant parameters from trained DNN models, thereby reducing the computational complexity and model size. Knowledge distillation techniques [7,8] are utilized to extract knowledge from large teacher models and distill it into smaller student models, thereby reducing the resource requirements for inference. The study in [18] employs knowledge distillation to deploy DNN models on resource-constrained UAVs for edge intelligence. By utilizing a full model trained in the cloud, it generates lightweight models with high accuracy. To enable the deployment of federated learning on edge devices, ref. [19] adopts a random pruning method to reduce communication overhead and model size, thereby improving the efficiency of federated learning on edge devices. However, these methods may lead to a decline in model performance, incurring losses in accuracy, precision, or other evaluation metrics. Additionally, some compression techniques introduce irregular sparsity, which can hinder effective hardware acceleration and reduce processing speed. As the primary focus in the aforementioned studies was training new lightweight networks to reduce resource demands, only homogeneous edge devices, which have limited adaptability, were considered. When different types of edge devices are introduced, these models must be retrained to fit the new device configurations; thus, they are unsuitable for heterogeneous edge-computing systems.

2.2. Cloud–Edge Collaboration

To reduce the burden of DNN inference on edge devices, the computationally intensive parts of the inference process are offloaded to cloud servers [9–11,20,21]. In [9], a full offloading approach, in which multiple variants of the DNN model were created, was proposed to facilitate the selection of different models based on the device's resource constraints; however, with limited resources, the inference performance of the selected model inevitably declined to some extent. In [10], a partial offloading solution was introduced to identify an intermediate partition point within the DNN structure; the initial layers were processed locally, whereas the subsequent layers were offloaded to the cloud. In [11], a method similar to that in [10] was proposed, focusing on mapping different components of the DNN to various layers in the cloud-edge hierarchy to reduce communication overhead and optimize resource usage. However, unlike our proposed method, the aforementioned approach required retraining the DNN model across different levels. In [20], a context-adaptive and dynamically composable DNN deployment framework was proposed. In this work, DNN models were pre-divided into DNN atoms, and during subsequent inference, offloading plans were adaptively formulated to reduce latency and save memory. The study in [21] explored how to achieve rapid DNN inference in UAV swarms through collaboration among multiple UAVs. The DNN model was first partitioned into multiple segments, with each UAV performing inference on a specific segment, and the results were then aggregated. Overall, these studies were focused on leveraging cloud servers to ease the inference load on edge devices. However, cloud–edge collaborative approaches exhibit bandwidth limitations in edge networks, leading to high round-trip latency and data privacy concerns. Therefore, cloud–edge collaborative methods are not suitable for edge-computing systems that demand strict real-time performance.

2.3. Distributed Inference

To further reduce the latency of inference tasks on edge devices, certain methods [12–17,22–24] are employed to divide DNN inference tasks into multiple independently executable subtasks. These subtasks can be executed in parallel across multiple

edge devices. In [12], a biased one-dimensional partitioning (BODP) method was proposed to divide the input of each network layer into strips and distribute more workload to devices with higher computational capacity. In [13], a greedy two-dimensional partitioning (GTDP) scheme was developed for layer partitioning; however, this approach significantly increased the synchronization overhead between devices with adjacent partitions. The methods in [12,13] executed distributed inference in a layer-by-layer manner, resulting in significant synchronization overhead between layers. Therefore, the methods in [12,13] are not well-suited for complex DNN models. In contrast, our approach avoids the layer-by-layer synchronization overhead, making it capable of handling models with a greater number of layers. In [14], the FTP method was proposed to vertically divide and fuse convolutional layers in a grid-like manner to reduce memory usage and communication overhead; however, this method is only applicable in scenarios with homogeneous devices. Based on [14], a fusion search strategy with dynamic programming that dynamically selects the optimal execution strategy based on the availability of computational resources and network conditions was introduced [17]. Unlike this work, the study in [14] does not consider scenarios involving heterogeneous edge devices. The study in [22] introduced DistrEdge, which leverages deep reinforcement learning to enable adaptability to various scenarios (e.g., different network conditions and diverse device types). DistrEdge identifies optimal split decisions for CNN models and executes inference tasks in parallel across edge devices to achieve accelerated inference. The study in [23] proposes SDPMP, which combines the principles of pipeline parallelism and the partial dependency characteristics within CNN layers to intelligently perform inter-layer and intra-layer partitioning, meeting the low latency requirements for single-task inference. While the studies in [17,22,23] all consider the heterogeneity of edge devices, unlike this work, they do not take energy consumption into account when reducing inference latency. Overall, the methods proposed in [12–17,22,23] focus on distributed inference across edge devices to achieve lower inference latency. Inspired by these works, we propose improvements to the FTP algorithm to make it suitable for the practical requirements of edge-computing systems. We explore DNN deployment in edge systems, including relevant models, algorithms, and methods for distributed DNN inference.

In summary, while prior works have made significant progress in distributed inference for edge systems, they often fail to consider energy consumption as a critical optimization objective, limiting their applicability in resource-constrained environments. Furthermore, most existing approaches lack effective handling of residual blocks in deep neural networks. To address these limitations, our work introduces a novel approach that incorporates energy consumption into the optimization framework while explicitly accounting for the unique challenges posed by residual blocks. Experimental results demonstrate that our method reduces energy consumption to a certain extent while also improving inference latency.

3. System Model and Problem Description

3.1. System Overview

Consider the edge-computing application illustrated in Figure 1, with N available edge devices represented as set $\mathcal{N} = \{U_1, U_2, ..., U_i, ..., U_N\}$. Each edge device runs the same DNN model, $\mathbb{G}$, consisting of L layers, excluding the fully connected layers, expressed as $\mathbb{G} = \{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_j, ..., \mathcal{L}_L\}$.

When an edge device initiates an inference task, it is designated as the master device (MD), whereas the others serve as worker devices (WDs). The general process of DNN inference in this situation proceeds as follows: the MD comprehensively considers the computational capabilities, as well as the computation and transmission power of heterogeneous edge devices. It assigns a greater workload to WDs with higher computational

capabilities and lower computation and transmission power. After the allocation, sub-tasks are sent to the WDs for parallel execution. The WDs then send their results back to the MD, which processes the data to complete the inference task.

The workload-partitioning decision made by the MD is represented as vector $\pi = \{a_1, a_2, ..., a_i, ..., a_N\}^T$, where $a_i \in Z^+$ denotes the portion of the workload allocated to edge device U_i. Vector π is considered to be an effective workload-partitioning decision if the conditions defined in Equations (1) and (2) are met:

$$a_i \geq 0, a_i \in \mathbb{Z}, \forall i \in N \tag{1}$$

$$\sum_{i \in N} a_i = H \tag{2}$$

Equation (1) requires that a_i must be a non-negative integer, and Equation (2) ensures that the sum of all elements in the partitioning strategy is equal to the height, H, of the output feature map produced by layer $\mathcal{L}_j$ of model $\mathbb{G}$.

The key symbols used throughout this paper are defined in Table 1.

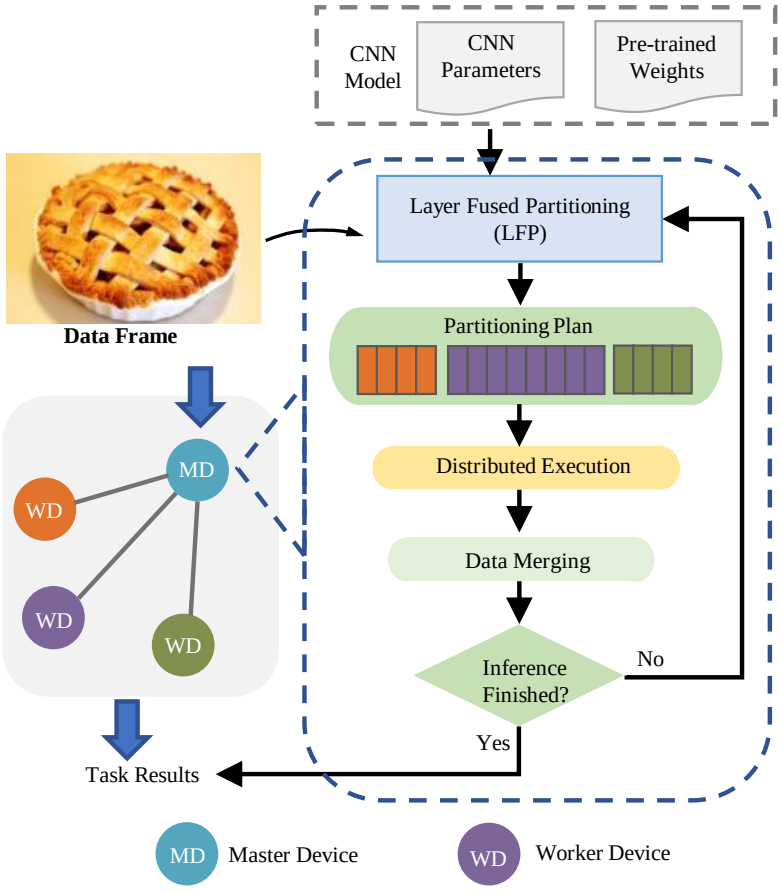


Figure 1. System overview.

Table 1. Variable symbols and their meanings.

Variable Symbol	Meaning	Variable Symbol	Meaning
U_i	Edge device number	C_{in}	The number of input channels for the $\mathcal{L}_j$'s input feature map
π	workload-partitioning decision	C_{out}	The number of output channels for the $\mathcal{L}_j$'s output feature map

Table 1. *Cont.*

Variable Symbol	Meaning	Variable Symbol	Meaning
$\mathcal{L}_j$	The j-th layer of the network in $\mathbb{G}$	$\mathbb{G}$	DNN model
H	The height of the output feature map of $\mathcal{L}_j$	$P_{comm,i}$	The computational power of U_i
W	The width of the output feature map of $\mathcal{L}_j$	$P_{comp,i}$	The transmission power of U_i
K	The kernel size of the convolutional layer $\mathcal{L}_j$	B	The bandwidth between U_i and U_j
α_i	The coefficients of the linear regression model of U_i	β_i	The coefficients of the linear regression model of U_i
p	The partition points of the network layer	μ_i	The proportion covered by the workload-partitioning decision of U_i

3.2. Estimation of Inference Latency and Energy Consumption of Edge Devices

3.2.1. Latency

When the MD assigns inference tasks to WDs, the latency primarily consists of two components: the computation latency on the WD, and the communication latency between the WD and MD. According to [13], under the given computational hardware and DNN model configuration, the computation time is approximately proportional to the number of floating-point operations (FLOPs) required by the layers of the DNN model. Additionally, as indicated in [25], the FLOPs, F_{ij}, for the convolutional kernel in network layer $\mathcal{L}_j$ on edge device U_i can be calculated by applying Equation (3):

$$F_{ij} = 2HW(C_{in}K^2 + 1)C_{out} \tag{3}$$

where H, W, and K represent the height, width, and kernel size of the input feature map for layer $\mathcal{L}_j$, respectively; and C_{in} and C_{out} denote the numbers of input and output channels in the feature map of $\mathcal{L}_j$, respectively.

(1) Computation Latency of WDs

The computation latency of the inference model on a WD is composed of the latency generated by two multi-layer fused networks. The computation time of a single DNN layer is approximated by using a linear regression model for convolutional computation time, $T^{pre}_{comp,ij}$ [16]. Typically, a higher computational capability of the edge device corresponds to a lower value of $T^{pre}_{comp,ij}$. Thus, the computation latency for layer $\mathcal{L}_j$ on edge device U_i is estimated as per Equation (4):

$$T^{pre}_{comp,ij} = \alpha_i F_{ij} + \beta_i \tag{4}$$

where α_i and β_i are parameters obtained from the linear regression model.

The latency generated by the two segments of the fused multi-layer network can be further estimated as follows. As the number of layers in the DNN increases, the inference precision increases; however, an excessively high number of layers leads to increased redundancy in the fused blocks. To address this, a partition point, p, is selected within the network layers to divide the multi-layer network into two segments, and the resultant fused blocks help minimize redundancy. Based on this partitioning strategy, the computation latency on the WD is estimated by using Equation (5):

$$T^{pre}_{comp,i} = \sum_{l_1} T^{pre}_{comp,il_1} + \sum_{l_2} T^{pre}_{comp,il_2} \tag{5}$$

where $l_1 \in [1, p], l_2 \in [p+1, L]$.

Equation (5) is used to split the DNN model into two fused blocks with reduced redundancy, accelerating the inference process.

The communication latency, $T^{pre}_{comm,ij}$, depends on the size of the feature maps being transmitted and the bandwidth, B, of the communication link between the MD and WD. After the fused block computation in layer $\mathcal{L}_j$ is completed, the communication latency, $T^{pre}_{comm,ij}$, required to transmit an output feature map having the dimensions of $C_{out} \times H \times W$ for layer $\mathcal{L}_j$ can be estimated using Equation (6) [16]:

$$T^{pre}_{comm,ij} = \frac{C_{out} \times H \times W \times 32}{1024 \times 1024 \times B} \tag{6}$$

where C_{out} represents the number of output channels in the feature map of layer $\mathcal{L}_j$, and H and W represent the height and width of the output feature map of $\mathcal{L}_j$, respectively.

Because the DNN network is divided into two fused blocks, the WD must synchronize computation results twice with the MD after completing its tasks. These synchronizations generate two communication overheads, corresponding to the synchronization of the results obtained from layers $\mathcal{L}_p$ and $\mathcal{L}_L$. Therefore, the transmission latency on the WD is calculated as per Equation (7):

$$T^{pre}_{comm,i} = T^{pre}_{comm,ip} + T^{pre}_{comm,iL} \tag{7}$$

3.2.2. Energy Consumption

The computation energy consumption on a WD is estimated using Equation (8):

$$E^{pre}_{comp,i} = P_{comp,i} T^{pre}_{comp,i} \tag{8}$$

where $P_{comp,i}$ represents the computational power of edge device U_i.

The communication energy consumption between the WD and the MD is calculated using Equation (9):

$$E^{pre}_{comm,i} = P_{comm,i} T^{pre}_{comm,i} \tag{9}$$

where $P_{comm,i}$ represents the transmission power of edge device U_i.

Based on this analysis, the total inference latency, T^{pre}, and total energy consumption, E^{pre}, for distributed DNN inference in the edge-computing system are calculated using Equations (10) and (11), respectively.

$$T^{pre} = \max_{i \in N} (T^{pre}_{comp,i} + T^{pre}_{comm,i}) \tag{10}$$

$$E^{pre} = \sum_{i \in N} (E^{pre}_{comp,i} + E^{pre}_{comm,i}) \tag{11}$$

3.3. Problem Description

(1) Problem Definition

In DNN-based distributed inference within an edge-computing system, latency determines the user experience, whereas energy consumption is a critical factor affecting user costs. Both latency and energy consumption are simultaneously considered to establish the following utility function based on the normalized weighted sum of these two factors:

$$Q_\pi = \lambda_t T^{pre} + \lambda_e E^{pre} \tag{12}$$

where $\pi = \{a_1, a_2, ..., a_i, ..., a_N\}^T$ represents the partitioning decision, and $\lambda_t, \lambda_e \in [0, 1]$, with $\lambda_t + \lambda_e = 1$, are the weights representing the influence of latency and energy consumption on the workload-partitioning decision.

Typically, the smaller the inference task assigned to edge device U_i, the smaller the response time of the WD and the corresponding energy consumption of U_i. Accordingly, the optimal combination of inference latency and energy consumption is modeled as the minimization utility function, as shown in Equation (13):

$$\arg\min_{a_i} Q_\pi \tag{13}$$

$$s.t. \begin{cases} C1: & a_i \geq 0, a_i \in \mathbb{Z}, \forall i \in N \\ C2: & \sum\limits_{i \in N} a_i = H \\ C3: & T^{pre} \leq \Delta_{\max} \end{cases} \tag{14}$$

where Q_π represents the minimized utility function, and a_i is the workload assigned to edge device U_i. Constraint C1 specifies that each a_i in the partitioning decision must be a positive integer. Constraint C2 ensures that the sum of all values in the partitioning decision equals height H. Constraint C3 defines the maximum allowable inference latency for the task.

Essentially, Equation (13) is an ILP problem with a large decision space, similar to other complex optimization problems.

3.4. Linear Programming Relaxation Algorithm

A linear programming problem is an optimization problem in which a linear objective function is subjected to constraints in the form of linear equations or inequalities. An ILP problem is a special case of linear programming in which all optimization variables are restricted to integer values [26]. The challenge in solving Equation (13) arises from the discrete nature of the integer variable (a_i). To efficiently generate feasible solutions, inspired by [15], we introduce a continuous variable, μ_i, to relax Equation (13). The relationship between μ_i and a_i is expressed in Equation (15):

$$a_i = \mu_i H, \forall i \in N \tag{15}$$

where H represents the height of the output feature map, and μ_i denotes the proportion of the partitioning decision covered by the i-th segment.

By combining Equations (1) and (2), we derive the constraints related to μ_i.

$$\mu_i \geq 0, \forall i \in N \tag{16}$$

$$\sum_{i \in N} \mu_i = 1 \tag{17}$$

Thus, Equation (13) can be transformed into the workload-partitioning optimization model expressed in Equation (18):

$$\arg\min_{\mu_i} Q_\pi \tag{18}$$

$$s.t. \begin{cases} C1: & \mu_i \geq 0, \forall i \in N \\ C2: & \sum\limits_{i \in N} \mu_i = 1 \\ C3: & T^{pre} \leq \Delta_{\max} \end{cases} \tag{19}$$

Using the relaxation of integer constraints makes Equation (18) a broader and more easily solvable version of the original problem in Equation (13). Solutions obtained from Equation (18) can then be used to indirectly derive the integer values of a_i.

3.5. Workload-Partitioning Method Based on Linear Programming Relaxation

To solve the optimization problem described in Equation (18), the problem expressed in Equation (13) first undergoes relaxation to enable efficient computation. First, the computation latency is estimated for each individual DNN layer. Subsequently, the computation latency across multiple layers, including the fused block computation latency, is calculated. Next, the transmission latency is estimated based on the size of the fused blocks, and the energy consumption is computed accordingly. Finally, the total latency and energy consumption are used to determine the optimal workload-partitioning strategy (π). Based on this approach, a workload-partitioning algorithm using the linear programming relaxation is formulated as described in Algorithm 1.

Algorithm 1 Workload Partition Algorithm (WPA)

Input: Edge Device: $\mathcal{N} = \{U_1, U_2, ..., U_i, ..., U_N\}$
DNN Model: $\mathbb{G} = \{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_j, ..., \mathcal{L}_L\}$
Configuration tuples: $\mathbb{C} = (K, C_{in}, C_{out}, S, P)_j, \forall j \in L$
Resources tuples: $(P_{comp}, P_{comm})_i, \forall i \in N$
Bandwidths:B
Dividing point:$p, p \in L$
Output: Partitioning decision:$\pi = \{\mu_1, \mu_2, ..., \mu_i, ..., \mu_N\}$

1: **Procedure** PARTITION($\mathcal{N}$)
2: Initializing the problem variables
3: Get the predict delay T^{pre} from Equation (10) //* Estimate the total inference latency
4: Get the predict energy consumption E^{pre} from Equation (11) //* Estimate the total inference energy consumption
5: Solve LP Problem Equation (18) to obtain π
6: **if** π satisfy Equation (19) **then**
7: **return** π
8: **else**
9: Find the minimum element μ_m in π
10: $\mathcal{N} \leftarrow \mathcal{N} - \{m\}$ //* Delete the edge deviceU_m
11: **return** PARTITION($\mathcal{N}$)
12: **end if**

The primary computational cost of Algorithm 1 arises from the recursive operation in line 11. However, the total number of recursive calls does not exceed N (i.e., the total number of edge devices). Therefore, the time complexity of Algorithm 1 is O(N).

4. Convolution-Layer Partitioning Algorithm

Previous studies [13,14] have shown that convolutional layers account for most of the computation time and memory usage in DNN inference. In [12], the convolutional layers were divided into independently executable segments to speed up inference across multiple edge devices. To minimize inference time and energy consumption and find the optimal mapping between edge devices and workload partitions, we develop an enhanced version of the FTP algorithm [14]. First, layers are partitioned along the height of the input feature map, with each segment assigned to a worker device. A split point (p) divides each network layer into two parts, forming two fused blocks to reduce redundancy. Additionally, ResNet residual blocks (BasicBlocks) are converted into convolutional layers to simplify distributed inference. The resultant algorithm is the one-dimensional partitioning algorithm termed as OD-FTP.

4.1. OD-FTP

(1) Height-Dimension Partitioning and Residual Blocks

Convolution-layer partitioning typically involves a two-dimensional grid-partitioning method for feature maps, as depicted in Figure 2a. In this approach, the feature map is divided into $N \times M$ partitions, with the workload distributed across $N \times M$ edge devices. However, this two-dimensional partitioning approach often results in computational redundancy. In this case, the workload of each edge device overlaps with those of three/four neighboring devices, leading to a decrease in inference performance. To overcome this limitation, we propose the height-dimension partitioning strategy illustrated in Figure 2b. In this approach, the feature map is divided into N partitions along the height, with each partition corresponding to the workload of one of the N edge devices. Because the feature map is partitioned along its height, the workload of each edge device creates computational redundancy with only one/two neighboring devices. This results in more efficient computation of the fused blocks.

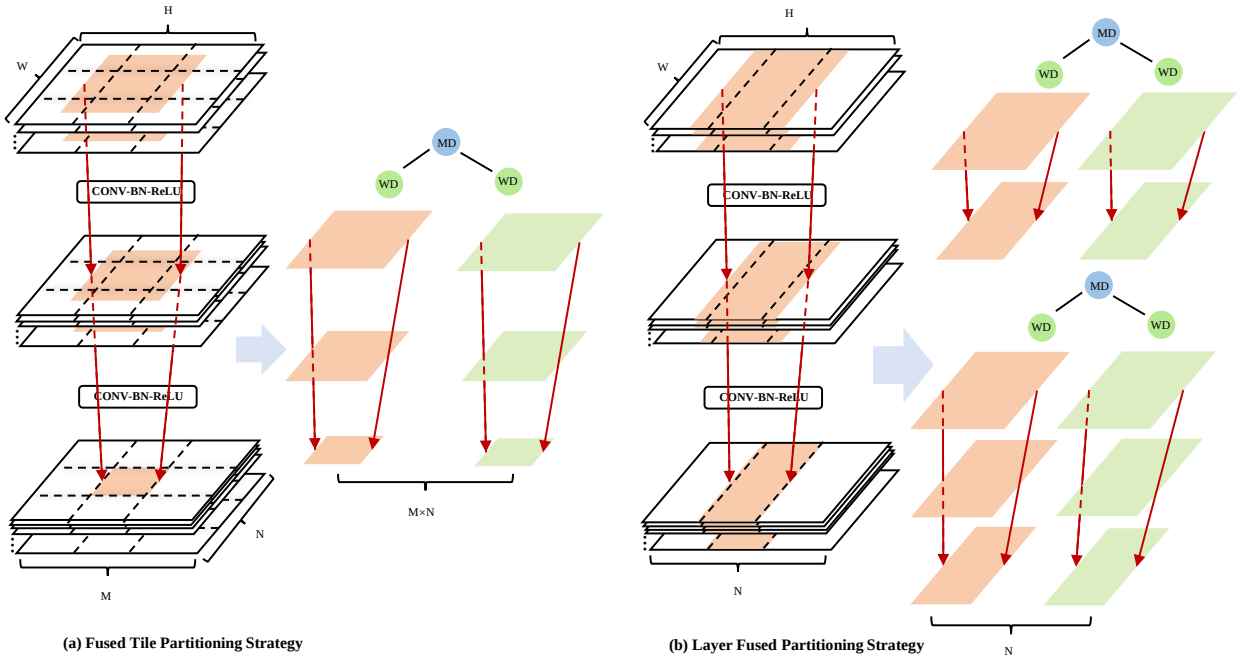
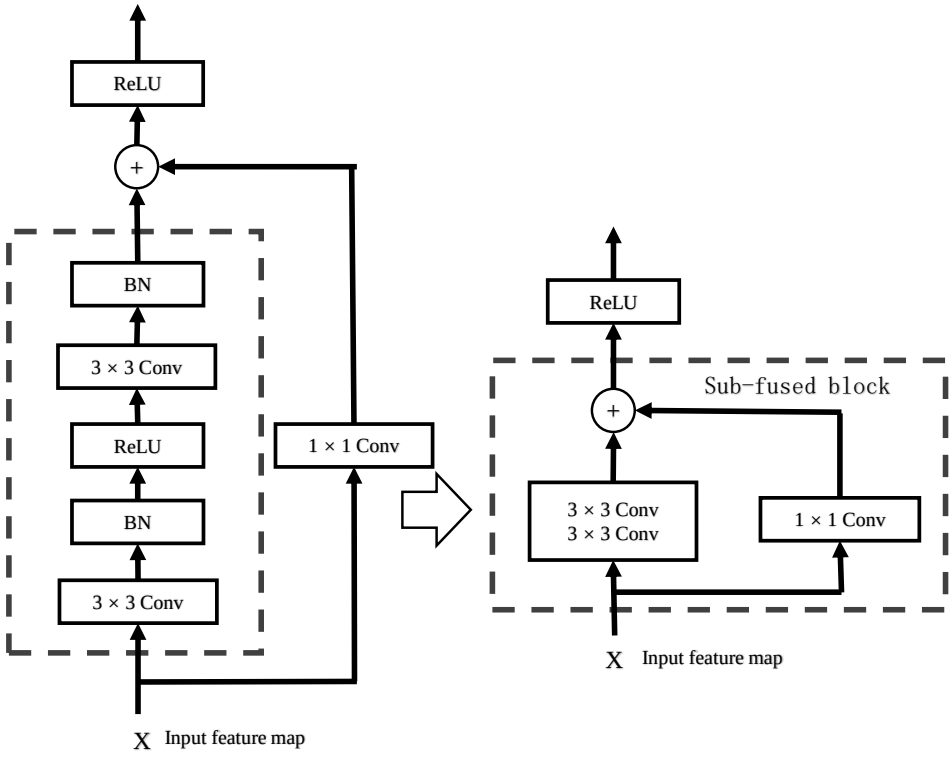


Figure 2. Different convolutional-layer partitioning strategies.

In the ResNet architecture, the execution order of network layers is structured as residual blocks instead of a simple sequential chain, as shown in Figure 3a. Residual blocks are essentially directed acyclic graphs (DAGs), where a layer can receive inputs from multiple preceding layers and provide outputs to several subsequent layers. Traditional distributed inference methods [12,14,15] require separately handling these DAG structures, which can degrade performance. To address this, we consider residual blocks as new convolutional layers, as displayed in Figure 3b. Converting residual blocks into sub-fused blocks avoids the complex layer dependencies within them and eliminates the need to handle each block individually. In this strategy, the ResNet model's execution order becomes a simple sequential model of layers. In the experiments conducted, the convolutional layers in ResNet18 and ResNet34 are transformed into 12- and 20-layer sequential networks, respectively.



(a) BasicBlock Struction **(b) Sub-fused block Struction**

Figure 3. BasicBlock structure transformed into sub-fused blocks.

(2) OD-FTP

Typical partitioning algorithms, such as the FTP algorithm, suffer from the issue of high computational redundancy within fused blocks. To address this issue, our proposed algorithm is used to calculate the transformation relationships between the input and output feature maps within the convolutional layers, and specifically, the padding size for each layer. After the partitioning strategy (π) is determined, the convolutional layer at partition point j on an edge device must compute a sub-intermediate feature map having the $C_j \times a_i \times W_j$ dimension and send the result to the MD. To validate this sub-intermediate feature map, the padding size for each convolutional layer within the fused block, denoted as (p_{up}, p_{down}), must be calculated based on the shape of the sub-intermediate feature map. Additionally, the relative position and length of the sub-intermediate feature map within the original feature map, represented as $(pos_{start}, length)$, must be determined. The concept underlying Algorithm 2 is calculating pos^{new}_{start} and pos_{end} through the stride of the convolutional layer and updating (p_{up}, p_{down}) by validating the pos^{new}_{start} and pos_{end} indices. The pseudocode of the proposed OD-FTP algorithm is presented in in Algorithm 2. The inputs for the OD-FTP algorithm include the configuration information related to the DNN layers and the shape of the original feature map after it passes through the current convolutional layer. The output consists of the relative position and length of the sub-intermediate feature map within the next convolutional layer and the padding size for that layer.

The main computational overhead of Algorithm 2 results from lines 2 and 3, and specifically from the computation of pos_{start} and $length$. The time complexity of Algorithm 2 is O(H).

Algorithm 2 OD-FTP

Input: DNN Layer: $\mathcal{L}_j, j \in L$
Configuration tuples: $\mathbb{C} = (K, C_{in}, C_{out}, S, P)_j, \forall j \in L$
Feature Map Shape: $(H, W)_j, H_j, W_j \in C_{in,j}$
Start Position: pos_{start}
Length: len
Output: Start Position: pos_{start}
Length: len
Padding: (p_{up}, p_{down})

Initializing the problem variables
Calculate the new location of the feature map pos_{start}^{new} //* Calculate the starting position of the partition decision solution
Calculates the new length of the feature map len^{new} //* Calculate the length of the partition decision solution
Calculates the end position of the feature map pos_{end}
if $pos_{start}^{new} < 0$ **then** //* Detect whether the subscript at the start position of the partition decision solution is out of bounds
    $len^{new} \leftarrow pos_{start}^{new} + len^{new}$
    $p_{up} \leftarrow -pos_{start}^{new}$ //* Calculate the top padding size for the network layer $\mathcal{L}_j$
    $pos_{start}^{new} \leftarrow 0$
end if
$pos_{\max} \leftarrow P_j + H_j - 1$ //* Calculate the maximum subscript
if $pos_{end} \geq pos_{\max}$ **then** //* Check if the subscript is out of bounds
    $len^{new} \leftarrow len^{new} - (pos_{end} - pos_{\max})$
    $p_{down} \leftarrow pos_{end} - pos_{\max}$ //* Calculate the bottom padding size for the network layer $\mathcal{L}_j$
end if
return $(pos_{start}, len^{new}, p_{up}, p_{down})$

4.2. Convolutional-Layer Partitioning Method Based on the OD-FTP Algorithm

The OD-FTP algorithm first obtains the optimal workload partitioning strategy, namely the "worker-device-to-workload" mapping, by invoking Algorithm 1. In this strategy, each worker device is assigned a portion of the original inference task. Subsequently, the OD-FTP algorithm computes the parallelizable partitions of the convolutional layers based on the workload. To enable distributed inference across heterogeneous edge devices, we integrated Algorithms 1 and 2 to formulate the LFP method. The pseudocode for the LFP method is presented in Algorithm 3. The objective of this method is to transform the workload-partitioning decision (π) into fused blocks that can be independently executed by edge devices. The LFP method first invokes Algorithm 1 to obtain the workload-partitioning decision (π). It then calls Algorithm 2, which calculates the $pos_{start}^{new}, len^{new}$ and padding (p_{up}, p_{down}) for the sub-feature maps within the relevant range of DNN layers, including convolutional and max-pooling layers. Subsequently, the LFP method extracts the sub-intermediate feature map, X_{sub}, from the original input feature map, X_{origin}. Finally, the MD sends X_{sub} and paddings to the WDs, which participate in distributed inference.

The computational overhead of the LFP method can be largely attributed to lines 5–14 of Algorithm 3. These lines correspond to the iterative calculation for the padding sizes of convolutional layers within fused blocks and the derivation of the position and length of the sub-intermediate feature map. Therefore, the time complexity of the LFP method is O(NH).

Algorithm 3 Layer Fused Partitioning Algorithm (LFP)

Input: Edge Device: $\mathcal{N} = \{U_1, U_2, ..., U_i, ..., U_N\}$
DNN Model: $\mathbb{G} = \{\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_j, ..., \mathcal{L}_L\}$
Configuration tuples: $\mathbb{C} = (K, C_{in}, C_{out}, S, P)_j, \forall j \in L$
Resources tuples: $(P_{comp}, P_{comm})_i, \forall i \in N$
Bandwidths:B
Dividing point:$p, p \in L$
Layer Fused Tile Scope: $(s, e), \forall s, e \in L$
Start Position: pos_{start}
Feature Map: X_{origin}
Output: Sub-feature Map:X_{sub}
Padding Configuration:$paddings$

$\pi \leftarrow WPA(\mathcal{N}, \mathbb{G}, \mathbb{C}, (P_{comp}, P_{comm}), B, p)$ //* Algorithm 1 is invoked to calculate the optimal workload partition decision
$a_i \leftarrow \pi$ //* Get the workload of the edge device U_i
$paddings \leftarrow []$ //* Stores the padding of multiple network layers
$(pos_{start}, len^{new}, p_{up}, p_{down}) \leftarrow (pos_{start}, a_i, 0, 0)$
for k = e -1 to s-1 **do**
  if $\mathcal{L}_k \in (Conv2d, Maxpool2d)$ **then**
    $(pos_{start}, len^{new}, p_{up}, p_{down}) \leftarrow OD - FTP(\mathcal{L}_k, \mathbb{C}_k, pos^{new}_{start}, len^{new})$ //* Algorithm 2 is called to calculate the relative position, length, and padding size
    add p_{up} to $paddings$
    add p_{down} to $paddings$
  else
    add 0 to $paddings$
    add 0 to $paddings$
  end if
end for
reverse $paddings$ list //*Since the padding size for each layer is calculated from back to front, the list needs to be reversed after the calculation
extract X_{sub} from X_{orgin} using pos^{new}_{start} and len^{new}
return $(X_{sub}, paddings)$ //* MD sends X_{sub} and $paddings$ to the edge device for parallel execution

5. Experimental Results and Analysis

This section presents simulation experiments conducted to verify the effectiveness and efficiency of the LFP method.

5.1. Evaluation Metrics

To evaluate the performance of the LFP method, three key metrics were used [15]: multiply–accumulate operations (MACs), latency, and energy consumption.

MACs: This metric is used to measure the computational complexity of neural networks. The calculation shown in Equation (20) represents the total number of basic operations (multiplications and additions) required during the forward propagation process. A lower MAC value typically indicates less computational load and higher efficiency. In this experiment, we used the thop library to calculate the MACs during DNN model inference on edge devices [27].

$$MACs = \mathrm{HW}(C_{in}K^2 + 1)C_{out} \tag{20}$$

Latency: Latency is the time required to complete a single inference task. In distributed inference systems, latency is usually determined based on the edge device that takes the longest time to complete its task. This metric is calculated using Equation (21).

$$T = \max_{i \in N}(T_{comp,i} + T_{comm,i}) \tag{21}$$

Energy consumption: This metric represents the total energy consumed by all edge devices after completing an inference task. Energy consumption is calculated as shown in Equation (22).

$$E = \sum_{i \in N} (E_{comp,i} + E_{comm,i}) \tag{22}$$

5.2. Experimental Setup

Considering that resource-constrained edge devices may not be equipped with GPUs, we implemented the proposed algorithm based on the PyTorch—CPU version. For the selection of edge devices, we created five virtual machines without GPUs on the BKYun cloud supercomputing platform [28] to simulate edge devices. We divided five virtual machines into three different types of edge devices to create a heterogeneous edge device environment. The virtual machines run the CentOS operating system, and their detailed configurations are listed in Table 2. Communication between the virtual machines is implemented using Python's socket API. For bandwidth control, we adopted the same strategy as described in [15], utilizing the traffic control tool TC (Traffic Control) to restrict the available bandwidth during the experiments.

Table 2. Virtual machine configuration in experiments.

Type	CPU	Memory	Number of Instances
V1	4 Core Intel Xeon Gold 6149 2.5 GHz (Intel, Santa Clara, CA, USA)	4 GB	2
V2	8 Core Intel Xeon (Skylake) Platinum 8163 2.5 GHz	8 GB	2
V3	8 Core Intel Xeon Platinum 8369 3.3 GHz	16 GB	1

In this experiment, four mainstream DNN models were implemented by using the PyTorch framework: VGG-16, VGG-19, ResNet-18, and ResNet-34. The workload consisted of an image classification task to be performed on the Food-101 dataset [29], with images sized at (512 × 512) pixels. Table 3. lists the main parameters used in the experiment. To eliminate random errors and improve the reliability of the experimental results, unnecessary programs were disabled during inference, and the inference latency was averaged out of 10 repeated experiments.

Table 3. Parameter settings for edge-device simulation experiments.

Parameter	Value	Definition
B	125	Total channel bandwidth (MB/s)
N	5	Number of edge device
$P_{comm,i}$	{10, 10, 12}	The computing power (W) of different types U_i
$P_{comp,i}$	{2, 2, 3}	The transmission power (W) of different types U_i
p	{17, 19, 6, 7}	The partition points for VGG16, VGG19, ResNet18, and ResNet34
λ_t, λ_e	(0.5, 0.5)	The weights of latency and energy consumption

5.3. Effectiveness of the LFP Method

The effectiveness of the LFP method was verified on the basis of the three evaluation metrics described in Section 5.1. To achieve a balanced optimization of both latency and energy consumption, equal weights were assigned to both metrics: $\lambda_t = \lambda_e = 0.5$.

Figure 4 illustrates the impact of not using distributed inference or the two-stage fusion block strategy on the performance of the LFP method across four models. Here, the Loc method represents local inference, while the NLF method indicates the absence of the two-stage fusion block strategy. In the figure, V1, V2, and V3 correspond to the edge devices listed in Table 1. For all four models, the proposed LFP method exhibits the lowest MACs, followed by the NLF method. This is because the LFP method partitions the convolutional layers of the DNN model into two segments, creating fusion blocks with lower computational redundancy. In contrast, the Loc method performs inference solely on a single edge device, which does not reduce the MACs.

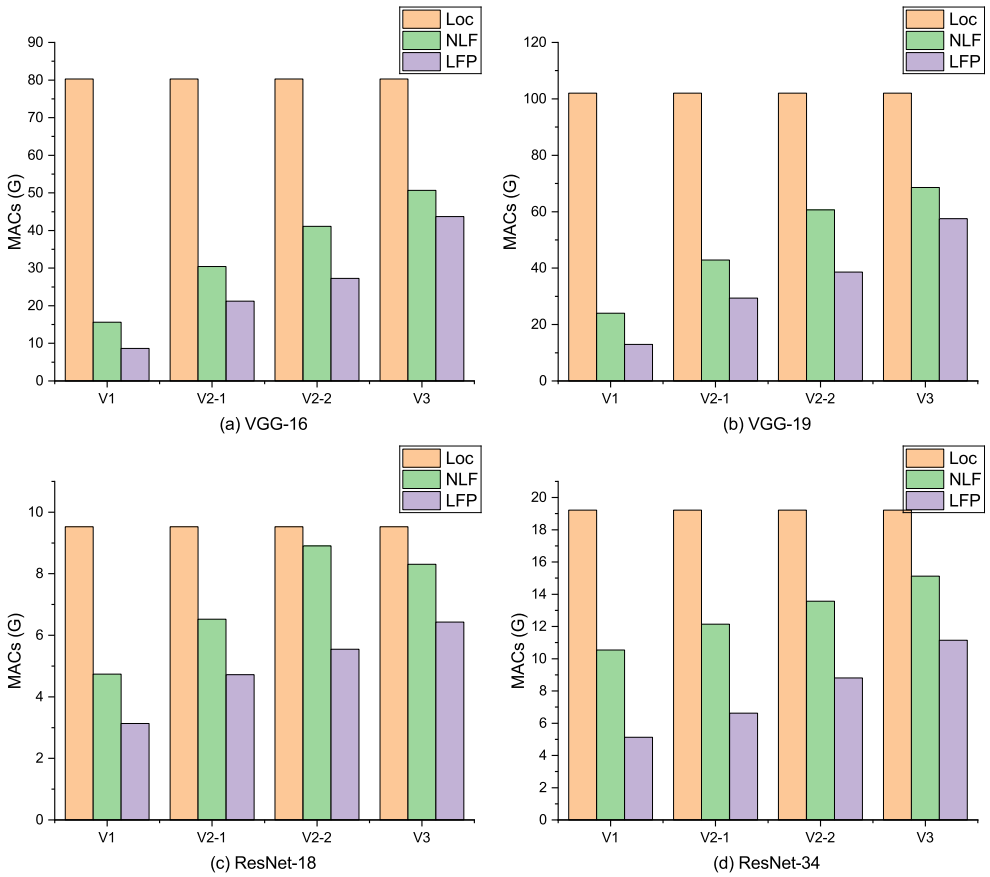


Figure 4. MAC metric.

The experimental results for the latency metric are shown in Figure 5. The proposed LFP method achieved the lowest latency across the four DNN models. This indicates that, according to the optimization model presented in this paper, the LFP method can derive the optimal workload partitioning strategy. This effectively utilizes other participating edge devices in distributed inference to accelerate the overall inference process.

The experimental results for the energy consumption metric are shown in Figure 6. The proposed LFP method exhibits the lowest energy consumption across the four DNN models. This is because the LFP method takes into account the computational capabilities and power characteristics of different edge devices. More powerful edge devices are assigned a greater workload, which helps reduce latency and indirectly lowers energy expenditure.

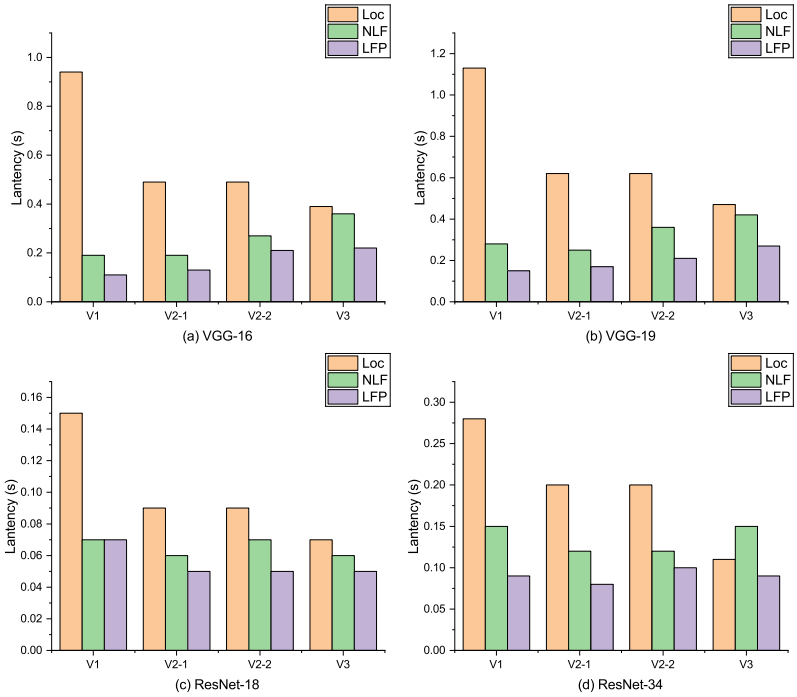


Figure 5. Latency metric.

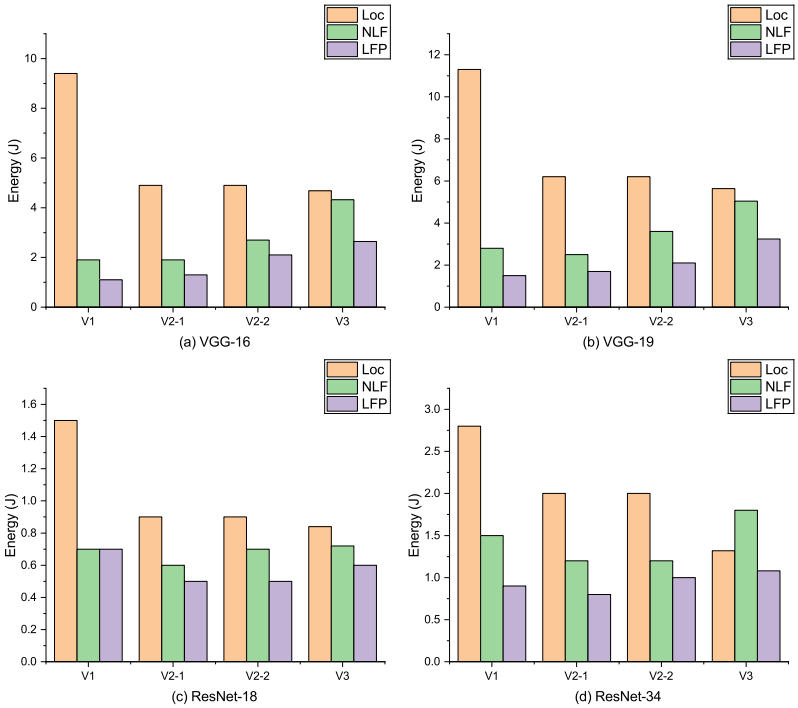


Figure 6. Energy metric.

5.4. Efficiency of the LFP Method

To evaluate the efficiency of the proposed LFP method, comparisons are made with BODP [12], FTP [14], CoEdge [15], EdgeFlow [24] and the Loc method in terms of MACs, latency, and energy consumption. Additionally, to ensure fairness, local inference (Loc) is set as the baseline, with its master device (MD) fixed as an edge device of type V1.

Figure 7 presents the latency results of different algorithms compared across four DNN models. As shown in Figure 7a,b, compared to local inference, the LFP method achieves 3.17× and 3.48× inference speedups on VGG-16 and VGG-19, respectively. The BODP, CoEdge, and EdgeFlow methods adopt a layer-wise synchronization strategy, which

results in longer inference times due to frequent synchronization of intermediate results. The BODP method fails to account for the heterogeneity of different edge devices, leading to imbalanced workload distribution and higher inference latency. Both the LFP and FTP methods utilize the fused block strategy to avoid frequent inter-layer synchronization. The LFP method divides the convolutional layers of the DNN model into two fused blocks with low computational redundancy, making it more efficient than the FTP method.

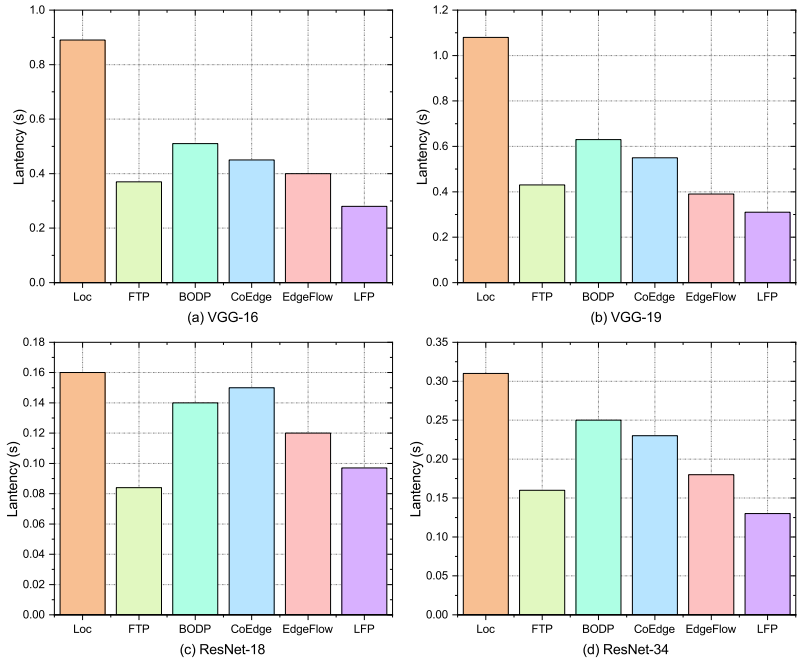


Figure 7. Comparison between the latencies of different methods across four DNN models.

As shown in Figure 7c,d, the LFP method achieved 1.65× and 2.38× inference speedups on ResNet-18 and ResNet-34, respectively. The BODP and CoEdge methods exhibit higher inference latency because they do not apply special processing to the residual blocks in the ResNet architecture, preventing optimal performance during distributed inference. In contrast, the LFP method converts residual blocks into sub-fused blocks, thereby eliminating the synchronization overhead within residual blocks. EdgeFlow encapsulates the layer dependencies of DAG-structured models into carefully partitioned execution units, which are executed in parallel.

As shown in Figure 7a,b, the speed of the LFP method becomes 3.17 times on VGG-16 and 3.48 times on VGG-19 compared with that of local inference. The LFP method divides the DNN model's convolutional layers into two fused blocks with low computational redundancy; thus, it is more efficient than the FTP method. By contrast, the FTP method groups all convolutional layers into a single fused block, which becomes overloaded with too many layers, thereby increasing redundancy and reducing efficiency. Furthermore, the BODP method suffers from high latency due to imbalanced workload partitioning and frequent inter-layer synchronization. As shown in Figure 7c,d, the speed of the LFP method becomes 1.65 times on ResNet-18 and 2.38 times on ResNet-34 compared with that of local inference. The LFP method converts the residual blocks in the ResNet model into sub-fused blocks and splits the convolutional layers into two smaller fused blocks, thereby reducing redundancy and improving efficiency.

Figure 8 presents the energy consumption experimental results for the comparative methods. As shown in Figure 8a,b, compared to local inference, the LFP method achieves energy savings of 16% and 20% on the VGG-16 and VGG-19 models, respectively. The FTP, BODP, and EdgeFlow methods exhibit higher energy consumption as they focus solely

on latency without considering energy consumption, leading to higher energy costs than local inference. Both CoEdge and LFP consume less energy than local inference, as they take into account the differences in computational and transmission power among edge devices during inference. As shown in Figure 8c,d, the LFP method is less energy-efficient for ResNet compared to local inference because the model's lower computational demand limits the optimization potential of the fused block strategy. The FTP, BODP, and EdgeFlow methods all consume more energy than local inference. This is because their optimizations prioritize latency in workload partitioning without considering the power consumption characteristics of different device types.

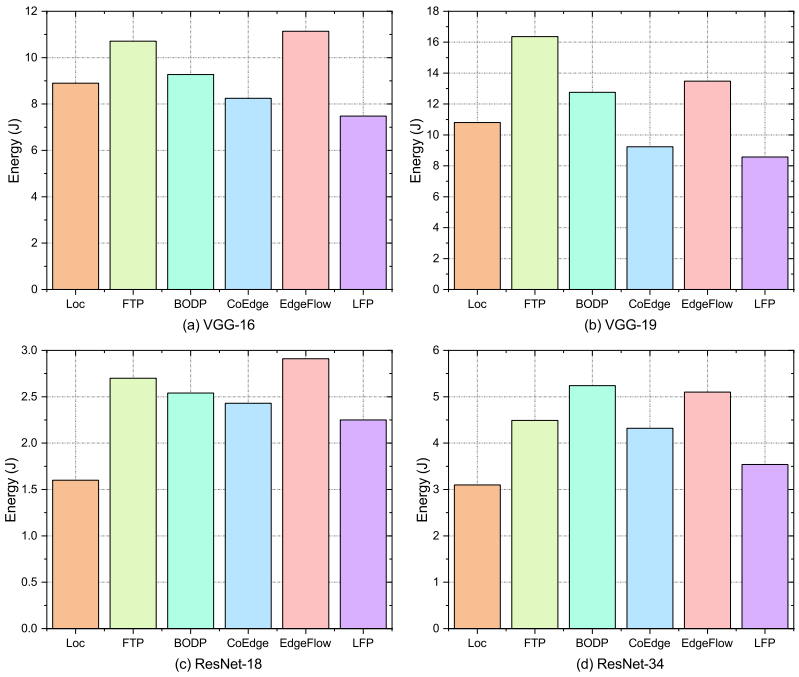


Figure 8. Comparison between the energy consumptions of different methods across four DNN models.

As shown in Figure 8a,b, the LFP method saves 16% and 20% energy, respectively, on the VGG-16 and VGG-19 models compared to local inference. This is because the LFP method accounts for differences in computational and transmission power between different edge devices. It utilizes the workload partitioning module proposed in Section 3.4 to achieve the optimal partitioning strategy. In contrast, the BODP and FTP methods focus solely on latency without considering energy consumption, resulting in higher energy costs than local inference. As depicted in Figure 8c,d, the LFP method is less effective than local inference for ResNet, as the model's lower computational demands limit the optimization potential of the fused block strategy. Both FTP and BODP exhibit higher energy expenditures than local inference. This is because their optimizations do not account for the power characteristics of different device types, prioritizing latency in workload partitioning instead. Overall, the LFP method achieves the lowest energy expenditure compared to FTP and BODP. This is accomplished by jointly optimizing computation and balancing the computational capabilities and power characteristics of devices.

In the experiments, we used 512 × 512 resolution images as input data for the DNN model. After completing the inference of the convolutional layers, the feature map shape becomes $C_{out} \times 16 \times 16$. According to the proposed partitioning algorithm with high partitioning, theoretically, up to 16 edge devices can be supported for distributed inference. However, this would result in significant communication overhead between the master device and the worker devices, leading to transmission latency far exceeding computation latency. As the computational capability of the edge devices increases, the computation

latency during distributed inference decreases. While the overall inference latency reduces, it is still partially affected by the transmission latency.

Figure 9 presents a comparison between the communication overheads of different algorithms across four edge devices. Both FTP and LFP use fused-block strategies to avoid data transmission between individual network layers, resulting in low communication overheads. However, LFP incurs a slightly higher overhead than FTP owing to the extra synchronization step that reduces redundancy between fused blocks. The BODP method, which involves layer-by-layer partitioning, incurs the highest communication overhead, considerably exceeding those of FTP and LFP. This can be attributed to the fact that the number of channels in the input feature map doubles every few layers during inference and the height and width shrink slowly. Consequently, each synchronization step transmits much larger amounts of data than the original inference task.

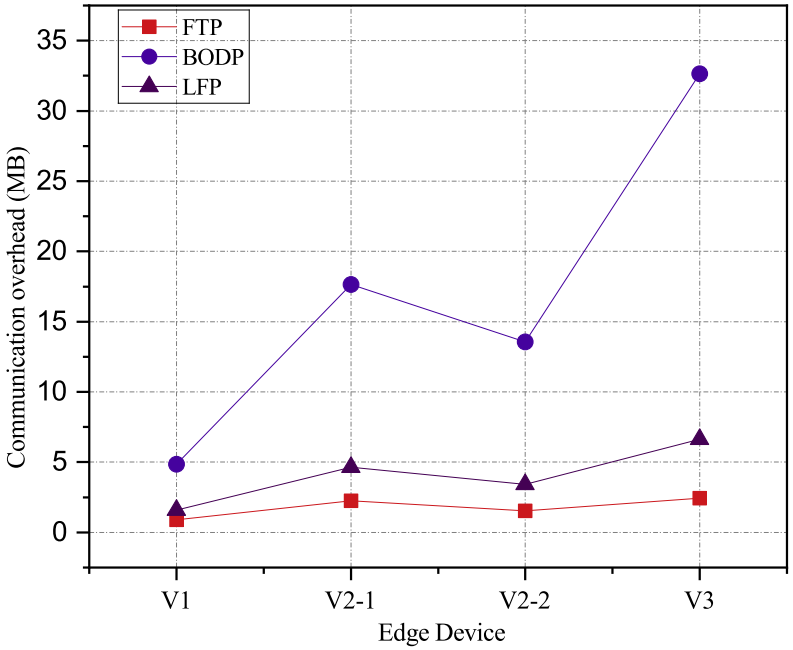


Figure 9. Comparison between the communication overheads of different algorithms on four edge devices.

To evaluate the robustness of the LFP method, we added new devices to the edge device cluster to measure latency and energy consumption. The DNN model selected for this experiment was VGG-19. Devices were added in the following sequence: V1, V1 and V2, V2, V3. Figure 10 presents the measurement results of the LFP method, with the devices added sequentially indicated at the top of the figure. As the number of devices increases, energy consumption initially decreases but then gradually rises because the computational energy consumption of the newly added devices exceeds the benefits gained from reduced latency. Latency gradually decreases as the number of devices grows, but the redundancy between fused blocks also increases. Eventually, latency stabilizes at a lower range and ceases to decrease further.

Figure 11 shows the changes in latency and energy consumption of the LFP method over time. In the experiment, we continuously ran the distributed inference system for one hour, performing an inference task every 10 min. The inference model was set to VGG19, with the primary device fixed as V1. The latency for each inference task remained stable at approximately 0.3 s, which is significantly lower than the local inference latency of V1 (1.3 s). Similarly, the energy consumption generated by the inference tasks followed the

latency trend, with the total energy consumption also lower than that of local inference (13 J).

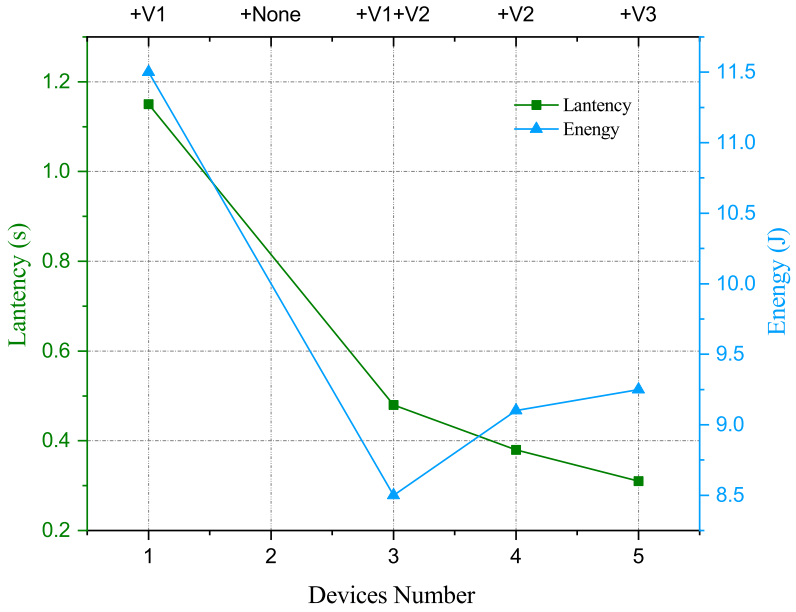


Figure 10. The latency and energy consumption of the LFP algorithm vary with the number of devices. The top text indicates which type of device are newly added to the cluster.

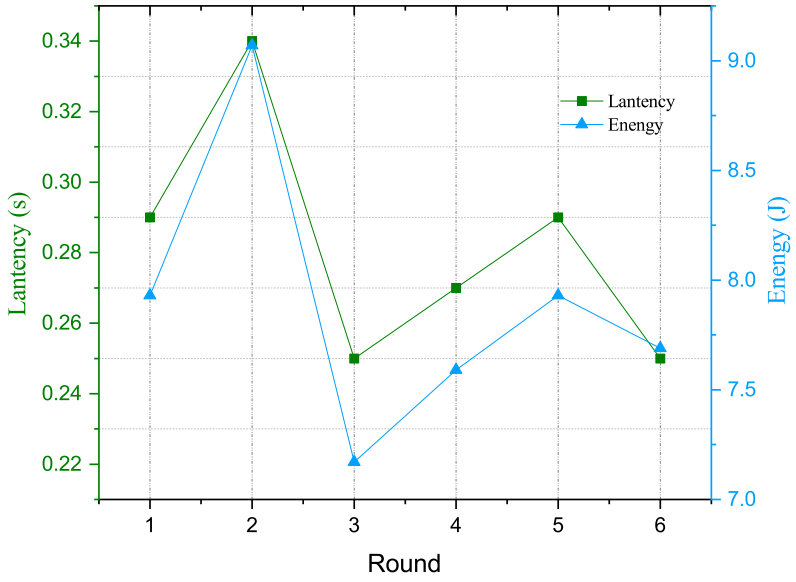


Figure 11. The latency and energy consumption of the LFP algorithm over time.

6. Discussion

The deployment of DNN models on resource-constrained heterogeneous edge devices is a critical challenge in edge computing, particularly in applications like smart cities, autonomous vehicles, and IoT systems. Efficient workload partitioning addresses the limitations of diverse edge devices, enabling faster inference and reduced energy consumption, which are essential for real-time, privacy-preserving applications.

In recent years, the Transformer architecture has been applied to various computer vision (CV) tasks with remarkable success, gradually replacing the classical CNN architecture. Therefore, our future work will focus on distributed inference for Vision Transformers models, aiming to provide more model options for inference on edge devices.

7. Conclusions

In this study, the deployment of DNN models on resource-constrained heterogeneous edge devices was investigated and a resource-adaptive workload-partitioning optimization model was proposed. The proposed model was used to address the challenge of determining workload allocation based on the heterogeneity of edge device resources. This remains an ongoing issue in current research. To efficiently solve the optimization model, a workload-partitioning algorithm was designed to determine effective partitioning strategies in real time. Based on this, an LFP algorithm was designed as an improved version of the FTP algorithm to further enhance inference performance on edge devices and minimize latency and energy consumption. Experimental results revealed that the proposed method increases the inference speed and reduces energy consumption compared with existent approaches across four widely used DNN models.

However, there are a few limitations that need to be further addressed in our future works. First, the proposed method lacks consideration of the partitioning for the non-convolutional layers, such as the fully connected layers. Second, the developed methods do not pay enough attention on the issue of edge device failures. Covering whole layers and the regarded factors to enhance the inference performance of the presented method will be the focus of our future works.

Author Contributions: Conceptualization, Q.Y. and Z.L.; methodology, Q.Y.; software, Q.Y.; writing—original draft preparation, Q.Y. and Z.L.; writing—review and editing, Q.Y. and Z.L.; supervision, Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: The authors did not receive any financial support for this study.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* **2019**, *107*, 1738–1762. [CrossRef]
2. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
3. Deng, L.; Li, G.; Han, S.; Shi, L.; Xie, Y. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc. IEEE* **2020**, *108*, 485–532. [CrossRef]
4. Yin, M.; Sui, Y.; Liao, S.; Yuan, B. Towards efficient tensor decomposition-based dnn model compression with optimization framework. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Nashville, TN, USA, 20–25 June 2021; pp. 10674–10683. [CrossRef]
5. Ruiquan, L.; Lu, Z.; Yuanyuan, L. Deep Neural Network Channel Pruning Compression Method for Filter Elasticity. *J. Comput. Eng. Appl.* **2024**, *60*, 163. [CrossRef]
6. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149. [CrossRef]
7. Blakeney, C.; Li, X.; Yan, Y.; Zong, Z. Parallel blockwise knowledge distillation for deep neural network compression. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *32*, 1765–1776. [CrossRef]
8. Hinton, G. Distilling the Knowledge in a Neural Network. *arXiv* **2015**, arXiv:1503.02531. [CrossRef]
9. Han, S.; Shen, H.; Philipose, M.; Agarwal, S.; Wolman, A.; Krishnamurthy, A. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, Singapore, 26–30 June 2016; pp. 123–136. [CrossRef]

10. Kang, Y.; Hauswald, J.; Gao, C.; Rovinski, A.; Mudge, T.; Mars, J.; Tang, L. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM Sigarch Comput. Archit. News* **2017**, *45*, 615–629. [CrossRef]
11. Teerapittayanon, S.; McDanel, B.; Kung, H.T. Distributed deep neural networks over the cloud, the edge and end devices. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 328–339. [CrossRef]
12. Mao, J.; Chen, X.; Nixon, K.W.; Krieger, C.; Chen, Y. Modnn: Local distributed mobile computing system for deep neural network. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1396–1401. [CrossRef]
13. Mao, J.; Yang, Z.; Wen, W.; Wu, C.; Song, L.; Nixon, K.W.; Chen, X.; Li, H.; Chen, Y. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 751–756. [CrossRef]
14. Zhao, Z.; Barijough, K.M.; Gerstlauer, A. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 2348–2359. [CrossRef]
15. Zeng, L.; Chen, X.; Zhou, Z.; Yang, L.; Zhang, J. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Trans. Netw.* **2020**, *29*, 595–608. [CrossRef]
16. Fang, W.; Xu, W.; Yu, C.; Xiong, N.N. Joint architecture design and workload partitioning for dnn inference on industrial iot clusters. *ACM Trans. Internet Technol.* **2023**, *23*, 1–21. [CrossRef]
17. Zhou, L.; Samavatian, M.H.; Bacha, A.; Majumdar, S.; Teodorescu, R. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, Washington, DC, USA, 7–9 November 2019; pp. 195–208. [CrossRef]
18. Luo, H.; Chen, T.; Li, X.; Li, S.; Zhang, C.; Zhao, G.; Liu, X. KeepEdge: A Knowledge Distillation Empowered Edge Intelligence Framework for Visual Assisted Positioning in UAV Delivery. *IEEE Trans. Mob. Comput.* **2023**, *22*, 4729–4741. [CrossRef]
19. Chen, C.; Jiang, B.; Liu, S.; Li, C.; Wu, C.; Yin, R. Efficient Federated Learning using Random Pruning in Resource-Constrained Edge Intelligence Networks. In Proceedings of the GLOBECOM 2023—2023 IEEE Global Communications Conference, Kuala Lumpur, Malaysia, 4–8 December 2023; pp. 5244–5249. [CrossRef]
20. Pang, B.; Liu, S.; Wang, H.; Guo, B.; Wang, Y.; Wang, H.; Sheng, Z.; Wang, Z.; Yu, Z. AdaMEC: Towards a Context-adaptive and Dynamically Combinable DNN Deployment Framework for Mobile Edge Computing. *ACM Trans. Sens. Netw.* **2023**, *20*, 1–28. [CrossRef]
21. Ren, W.; Qu, Y.; Qin, Z.; Dong, C.; Zhou, F.; Zhang, L.; Wu, Q. Efficient Pipeline Collaborative DNN Inference in Resource-Constrained UAV Swarm. In Proceedings of the 2024 IEEE Wireless Communications and Networking Conference (WCNC), Dubai, United Arab Emirates, 21–24 April 2024; pp. 1–6. [CrossRef]
22. Hou, X.; Guan, Y.; Han, T.; Zhang, N. DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices. In Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 30 May–3 June 2022; pp. 1097–1107. [CrossRef]
23. Han, B.; Dai, P.; Li, K.; Zhao, K.; Lei, X. SDPMP: Inference Acceleration of CNN Models in Heterogeneous Edge Environment. In Proceedings of the 2024 7th World Conference on Computing and Communication Technologies (WCCCT), Chengdu, China, 12–14 April 2024; pp. 194–198. [CrossRef]
24. Hu, C.; Li, B. Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices. In Proceedings of the IEEE INFOCOM 2022—IEEE Conference on Computer Communications, Virtual, 2–5 May 2022; pp. 330–339. [CrossRef]
25. Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J. Pruning convolutional neural networks for resource efficient inference. *arXiv* **2016**, arXiv:1611.06440. [CrossRef]
26. Dantzig, G.B. Linear programming and extensions. In *Linear Programming and Extensions*; Princeton University Press: Princeton, NJ, USA, 2016. [CrossRef]
27. Lyken17. Pytorch-OpCounter. 2020. Available online: https://github.com/Lyken17/pytorch-OpCounter (accessed on 11 December 2023) .
28. Shenzhen Beikun Cloud Computing Co., Ltd. Bei Kunyun Supercomputing Platform. 2019. Available online: https://www.bkunyun.com/ (accessed on 11 September 2023).
29. Bossard, L.; Guillaumin, M.; Van Gool, L. Food-101–mining discriminative components with random forests. In Proceedings of the Computer vision–ECCV 2014: 13th European conference, Zurich, Switzerland, 6–12 September 2014; Proceedings, Part VI 13; Springer: Berlin/Heidelberg, Germany, 2014; pp. 446–461. [CrossRef]