Towards Real-time Cooperative Deep Inference over the Cloud and Edge End Devices

SHIGENG ZHANG* and YINGGANG LI, Central South University, P. R. China
XUAN LIU†, Hunan University, P. R. China
SONG GUO, Hong Kong Polytechnic University, China
WEIPING WANG, Central South University, China
JIANXIN WANG, Central South University, China
BO DING, National University of Defense Technology, China
DI WU, Hunan University, China

Deep neural networks (DNNs) have been widely used in many intelligent applications such as object recognition and automatic driving due to their superior performance in conducting inference tasks. However, DNN models are usually heavyweight in computation, hindering their utilization on the resource-constraint Internet of Things (IoT) end devices. To this end, cooperative deep inference is proposed, in which a DNN model is adaptively partitioned into two parts and different parts are executed on different devices (cloud or edge end devices) to minimize the total inference latency. One important issue is thus to find the optimal partition of the deep model subject to network dynamics in a real-time manner. In this paper, we formulate the optimal DNN partition as a min-cut problem in a directed acyclic graph (DAG) specially derived from the DNN and propose a novel two-stage approach named *quick deep model partition (QDMP)* to solve it. QDMP exploits the fact that the optimal partition of a DNN model must be between two adjacent cut vertices in the corresponding DAG. It first identifies the two cut vertices and considers only the subgraph in between when calculating the min-cut. QDMP can find the optimal model partition with response time less than *300ms* even for large DNN models containing hundreds of layers (up to 66.3x faster than the state-of-the-art solution), and thus enables real-time cooperative deep inference over the cloud and edge end devices. Moreover, we observe one important fact that is ignored in all previous works: As many deep learning frameworks optimize the execution of DNN models, *the execution latency of a series of layers in a DNN does not equal to the summation of each layer's independent execution latency*. This results in inaccurate inference latency estimation in existing works. We propose a new execution latency measurement method, with which the inference latency can be accurately estimated in practice. We implement QDMP on real hardware and use a real-world self-driving car video dataset to evaluate its performance. Experimental results show that QDMP outperforms the state-of-the-art solution, reducing inference latency by up to 1.69x and increasing throughput by up to 3.81x.

*Shigeng Zhang is also with the the State Key Laboratory for Novel Software Technology, Nanjing University, P.R China.
†Xuan Liu is the corresponding author of this paper. Dr. Xuan Liu is also with the Science and Technology on Parallel and Distributed Processing Laboratory (PDL).

Authors' addresses: Shigeng Zhang, sgzhang@csu.edu.cn; Yinggang Li, liyinggang@csu.edu.cn, Central South University, Changsha, P. R. China; Xuan Liu, xuan_liu@hnu.edu.cn, Hunan University, Changsha, P. R. China; Song Guo, song.gong@polyu.edu.hk, Hong Kong Polytechnic University, Hong Kong, China; Weiping Wang, wpwang@csu.edu.cn, Central South University, Changsha, China; Jianxin Wang, jxwang@csu.edu.cn, Central South University, Changsha, China; Bo Ding, bding@msn.com, National University of Defense Technology, Changsha, China; Di Wu, dwu@hnu.edu.cn, Hunan University, Changsha, China.

2474-9567/2020/6-ART69 $15.00
https://doi.org/10.1145/3397315

CCS Concepts: • **Computer systems organization** → *Real-time systems*; • **Networks** → *Network protocols.*

Additional Key Words and Phrases: Edge AI, deep model partitioning, deep model acceleration, cooperative deep inference

ACM Reference Format:
Shigeng Zhang, Yinggang Li, Xuan Liu, Song Guo, Weiping Wang, Jianxin Wang, Bo Ding, and Di Wu. 2020. Towards Real-time Cooperative Deep Inference over the Cloud and Edge End Devices. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 2, Article 69 (June 2020), 24 pages. https://doi.org/10.1145/3397315

1 INTRODUCTION

The explosive progress of deep learning [15] has greatly improved the accuracy of many inference tasks that previously required human efforts, such as computer vision, speech recognition, and video analysis. Combined with the development of massive data processing techniques like cloud computing, deep learning has stimulated many intelligent applications in a variety of fields, e.g., automatic driving and augmented reality [3]. Generally, in these applications, the end devices generate a large volume of data and transmit the data to a cloud platform, where task-specific deep neural network (DNN) models are executed to analyze the data and the results are transmitted back to the end devices. For example, in automatic driving, the in-vehicle camera continuously monitors surrounding scenes and streams the video to the edge server, which performs scene recognition and target detection to assist automatic controlling of the vehicle.

This cloud-centric computing paradigm usually incurs a large latency because the end devices need to transmit a large volume of data to the cloud or the edge server, especially when the network status is poor [18]. For example, the autopilot camera captures more than 750 MB video data per second [29], which is far beyond the uplink bandwidth current transmission technologies can provide. Moreover, there might also be privacy concerns when transmitting all raw data to the cloud, especially for many privacy-sensitive applications like smart health. On the other hand, because DNN models are usually heavyweight and computation-intensive, they are not suitable to be deployed and executed on the resource-constraint Internet of Things (IoT) end devices. For example, the VGG model [23] requires approximately 100 MB memory and takes more than 2 seconds to infer one image on the Raspberry Pi 3B platform, which is too slow to support real-time inference.

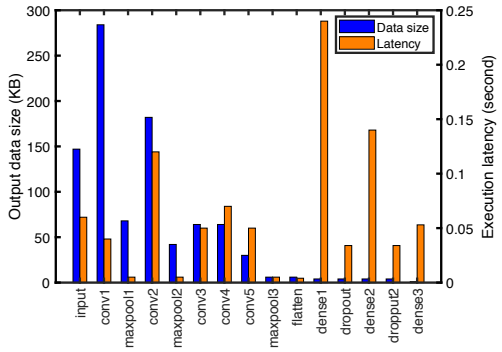


Fig. 1. Output data size and execution latency of each layer in AlexNet.

Cooperative deep inference [5, 9, 12, 30, 37] is proposed in recent years to take advantage of both the high computation ability of the cloud and the local processing ability of edge end devices. As shown in Figure 1, the execution latency and output data size of each layer in a DNN model (AlexNet in Figure 1) diverges significantly. For example, for layers in the front of the model, the output data sizes are usually large but the processing latency is usually small. In contrast, for layers in the tail of the model, the out data sizes are very small but the processing

latency is large. This motivates us to execute some front layers on edge end devices to reduce the volume of data to be transmitted and execute the tail layers on the cloud server to leverage its high computation ability, aiming to reduce the total inference latency. In order words, in cooperative deep inference, we partition the DNN model into two parts and execute different parts on different devices (cloud or end devices). The objective is to find the optimal partition of the DNN model to minimize the total inference and transmission latency subject to the current network status.

The model partition process needs to be time-efficient due to two reasons. First, because the partition decision is made on end devices that are usually constraint in computational ability, the partition process should be lightweight in computation; otherwise, it might take a very long time or sometimes even fail to obtain correct partition for large DNN models that might contain hundreds of layers. Second, when the network status (e.g., bandwidth) changes, the optimal partition of a given DNN model also changes. In order to quickly adapt to the dynamic network status and always use the optimal partition, it is critical to quickly re-compute the optimal partition and switch to the new partition after detecting network status changes, especially for tasks that require real-time response. In fact, if we use an incorrect partition after the network status changes, it might significantly increase the inference latency as depicted in existing works [9, 12] and our experiments.

Although there are few works on this topic [9, 12], however, they fail to provide a time-efficient optimal partition of general DNN models between the cloud and edge end devices, especially when the model is large. Neurosurgeon [12] initiates the research on optimal DNN model partition, but it handles only chain topological DNN models. Most recent DNN models (e.g., the widely used GoogleNet and ResNet [7]) have directional acyclic graph (DAG) topology, but Neurosurgeon cannot correctly partition DAG models as pointed out in [9]. The DADS approach proposed in [9] can correctly partition DNN models with DAG topology, but its time complexity is too high to achieve real-time partition decision on end devices. For example, our experiments show that it takes 68.42 seconds to partition GoogleNet on a Raspberry Pi 3B platform, too slow to react to the network dynamics in real-time. Moreover, DADS might fail to find proper partition for DAG topological DNNs in some practical settings.

In this paper, we propose the *quick deep model partition (QDMP)* approach that can partition a large DNN model with hundreds of layers in *second-level* time, and thus enables real-time cooperative deep inference over the cloud and edge end devices. We formulate the optimal DNN model partition as a min-cut problem in a DAG specially derived from the DNN and propose a time-efficient two-stage algorithm to solve it. QDMP is inspired by the observation that the optimal partition of a DNN model must be between two adjacent cut vertices in the corresponding DAG. It first identifies the two cut vertices and considers only the subgraph in between when calculating the min-cut. QDMP achieves up to 66x speedup in finding the optimal partition of a DNN model when compared with the state-of-the-art solution. Moreover, we observe one important fact that is ignored in all previous works: As many deep learning frameworks optimize the execution of DNN models, *the execution latency of a series of layers in a DNN does not equal to the summation of each layer's independent execution latency*. Previous works all use per layer independent execution latency to calculate the optimal partition, which results in inaccurate inference latency estimation. We propose a new execution latency measurement method, with which the inference latency can be accurately estimated in practice.

In summary, we make the following major contributions in this paper:

- A novel two-stage approach named QDMP is proposed to find the optimal DNN partition. QDMP significantly reduces time complexity in finding the optimal partition and achieves second-level decision time even for large DNNs containing hundreds of layers, up to 66x faster than the state-of-the-art solution.
- A new method to construct a proper DAG for a given DNN model is developed, with which the optimal partition problem can be formulated as a min-cut problem. We show that the DAG constructing method

Table 1. Decision time (seconds) for typical DNN models on different hardware platforms.

DNN model	#Layer	Raspberry Pi 3B	Raspberry Pi 4B	Intel i3-3240
Tiny-YOLO	16	0.10	0.03	0.01
AlexNet	24	0.14	0.05	0.01
AlexNet-Parallel	28	0.16	0.06	0.02
DarkNet19	27	0.17	0.05	0.02
ResNet-18	111	18.14	4.88	1.88
GoogLeNet	166	68.42	18.18	6.82

described in [9] is flawed and might fail to find the correct partition under practical settings. The new graph construction method proposed in this paper fixes the problem.
- A new latency measurement method to obtain per layer execution latency is developed, with which the total inference latency can be accurately estimated in practice.
- QDMP is implemented on hardware and evaluated with a real-world self-driving car video dataset. The results demonstrate up to 1.61x improvement in latency and 3.81x improvement in throughput when compared with the state-of-the-art solution.

The rest of the paper is organized as follows. In Section 2 we report some preliminary results that motivate the design of QDMP. In Section 3 we describe how to construct proper DAG for a given DNN model, with which the optimal DNN partition problem can be formulated as a min-cut problem. We also discuss how to measure correct per layer latency in Section 3. In Section 4 we present our two-stage QDMP approach and analyze its time complexity. Experimental results and comparisons with the state-of-the-art solution are given in Section 5. We discuss the limitations of our work and the potential future directions it may open in Section 6. Related works are discussed in Section 7. Finally, concluding remarks are given in Section 8.

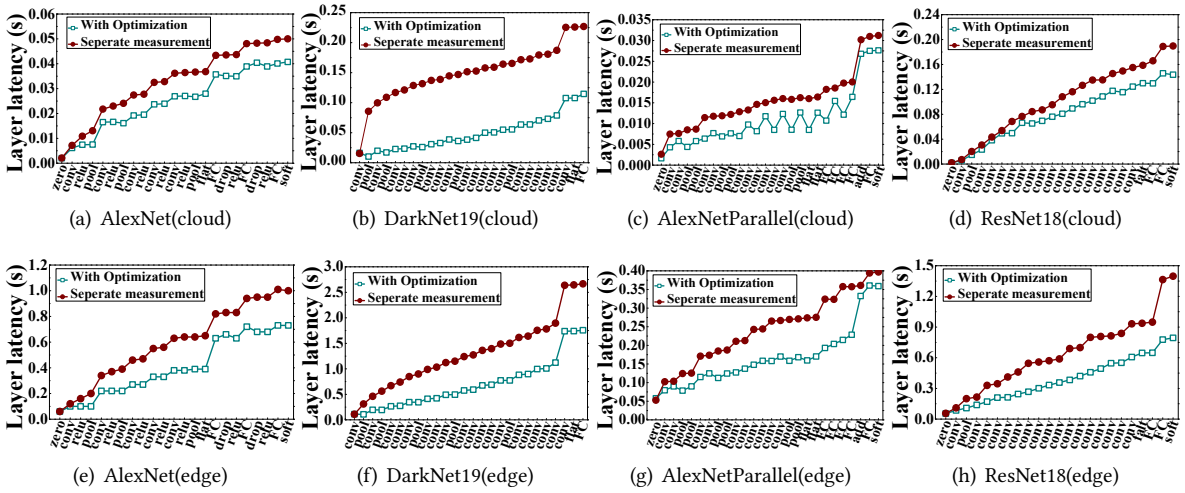
2 PRELIMINARIES

In this section, we present some preliminary experimental results that motivate the design of QDMP.

2.1 Decision Time

Considering that the optimal DNN model partition is decided at resource-constraint IoT end devices, the decision time to find the optimal partition should be short to support real-time adaption to network dynamics. We implement the state-of-the-art DADS [9] algorithm on three typical end device platforms (Raspberry Pi 3B/4B and Intel i3-3240) and list its decision time for some typical DNN models in Table 1. It can be observed that the decision time significantly increases when the number of layers in the model increases and the model becomes more complex. For example, on the Raspberry Pi 3B platform which is a typical platform to design IoT end devices, it takes more than 18.14 seconds to partition an 111-layer ResNet-18 model. For GoogLeNet which has 166 layers, the decision time is even longer than one minute. Even on a personal computer equipped with Intel i3-3240 CPU@3.4GHz, it still takes nearly 7 seconds to find the optimal partition for GoogLeNet.

The results in Table 1 show that the decision time of existing works is too long to support real-time adaption to network dynamics. As shown in existing works [9, 12] and validated in our experiments, the inference latency might significantly increase if a model is improperly partitioned and executed. Considering that the optimal partition is different under different network status, we need to quickly find the optimal model partition and switch to it after the network status changes. Moreover, because the DNN models are becoming more complex (e.g., containing more layers), it is necessary to find time-efficient approaches to partitioning large models to support real-time cooperative deep inferences.



(a) AlexNet(cloud) (b) DarkNet19(cloud) (c) AlexNetParallel(cloud) (d) ResNet18(cloud)

(e) AlexNet(edge) (f) DarkNet19(edge) (g) AlexNetParallel(edge) (h) ResNet18(edge)

Fig. 2. Cumulative execution latency of four typical DNN models at different layer on the cloud and edge end devices, where values marked with "With optimization" are measured considering continuous execution of a series of layers and values marked with "Separate measurement" are measured by feeding random values to each layer and execute that layer independently. The former is the actual latency, while latter is adopted in previous works such as [9, 12]. These models are implemented with the Caffe framework.

2.2 Per Layer Execution Latency Measurement

One important step in DNN model partition is to accurately measure per layer execution latency in the model. In previous works [6, 9, 12], the per layer latency is measured independently for each layer. For each layer, some random inputs are generated and fed into a layer and the execution time on the random input is used as the execution latency of that layer. However, many deep learning frameworks (e.g., Caffe [11], PyTorch [20], and Tensorflow [1]) will optimize the execution of DNN models, making *the execution latency of a series of layers in a DNN different from the summation of each layer's independent execution latency*. For example, in the Caffe framework, the `linear activation` layer sets the outputs of the convolution layer smaller than 0 to 0, and thus reducing matrix operations in the following layers. This will decrease the execution latency of the next layer and also affect all the following layers.

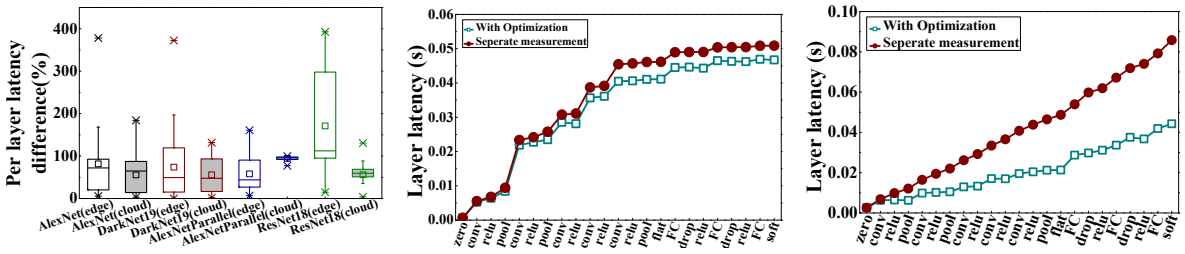
Our experimental results validate this fact. Taking the Caffe framework as an example, we measure the accumulative layer execution latency for four typical DNN models when considering each layer independently and when considering potential cross-layer optimization (the details can be found in Section 3, and plot the results in Figure 2. It can be observed that the execution latency measured with different methods differs significantly, on both the cloud and the edge end devices. In most cases, the latency measured in the separate mode is significantly larger than that measured in the optimization mode, which is the actual latency. The difference becomes more significant for the tail layers in the DNN model. Moreover, it can be observed that, with framework optimization, the accumulative execution latency of a front layer might even be smaller than that of the following layer, as shown in Figure 2(c) and Figure 2(e). This means traditional layer latency measurement approaches may fail to obtain correct per layer latency and hence might result in an improper model partition.

It can also be observed that for different models the accumulative layer differences are also diverse. In Figure 3(a) we plot the relative per layer latency difference in the two different measurement methods. It can be observed that,

for all the eight tested cases, the median relative per layer latency difference is higher than 50%. The maximum per layer latency difference in all the eight cases is higher than 150% except for AlexNetParallel and ResNet18 at the cloud side. The ResNet18 model at the edge device shows the most significant per layer latency difference, with the mean per layer latency difference approaches 200% and the maximum difference of nearly 300%.

We should point out that the phenomenon discussed in this section is a common phenomenon that can also be found in other deep learning frameworks such as PyTorch [20] and TensorFlow [1]. To validate this point, we implement AlexNet with PyTorch and TensorFlow and measure the per layer execution latency in different modes at the cloud side. The results are plotted in Figure 3(b) and Figure 3(c). Apparent differences in per layer execution latency can be observed for both frameworks, and the differences in the TensorFlow framework is more significant than in that the PyTorch framework. These observations are accordant with the results reported in a recent work [6].

These facts motivate us to develop new methods to measure per layer latency in DNN models, which is critical to find the optimal partition of the DNN model.



(a) Layer latency difference (b) PyTorch per layer latency (c) TensorFlow per layer latency

Fig. 3. (a) Relative per layer latency distribution difference for different DNN models, (b) per layer latency of AlexNet with PyTorch, and (c) per layer latency of AlexNet with TensorFlow.

3 PROBLEM FORMULATION

3.1 Modeling A DNN as a DAG

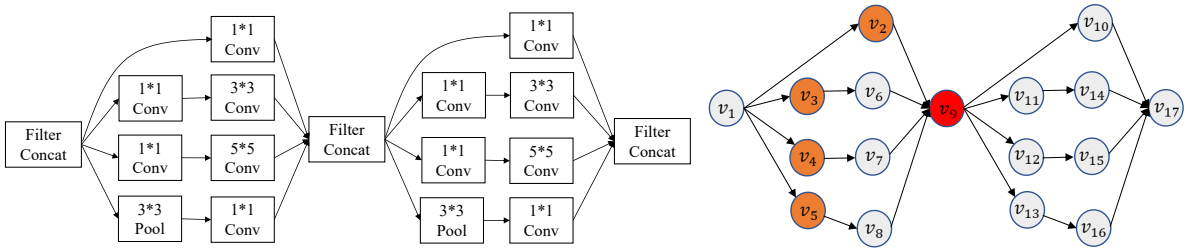
DNN models usually contain multiple layers, and each layer has one or more input and one output. The output of one layer is fed into the following layer(s) as (part of) the input of these layers. Because the data flow only in one direction[1], we can use a *directed acyclic graph* to represent a DNN model. For example, Figure 4(a) shows a piece of the GoogleNet [24], which can be modeled as a DAG as shown in Figure 4(b).

Generally, given a DNN model $\mathbf{M}$, we construct a DAG $\mathbf{G} =< V, E >$ to represent $\mathbf{M}$ as follows. Each vertex $v_i \in V$ corresponds to one layer in $\mathbf{M}$. There is one directed edge $< v_i, v_j > \in E$ if and only if the output data of layer corresponding to v_i are transferred to the layer corresponding to v_j as input. Note that each vertex might have multiple edges starting from it and multiple edges ending at it. For example, v_9 in Figure 4(b) (corresponding to the `Filter Concat` layer in Figure 4(b)) has four edges starting from it and four edges ending at it.

3.2 Optimal DNN Partition as a Min-Cut

There are three types of latency to be considered in a DNN model. Because each layer can be executed either in the cloud or on the edge end device, each layer has two latencies. We use t_i^c to denote the execution latency of

[1]In this paper, we do not consider DNN models with loops such as LSTM neural networks [31].



(a) A piece of GoogleNet (b) Corresponding DAG

Fig. 4. A piece of GoogLeNet [24] (a) and the DAG constructed from the piece (b).

v_i's corresponding layer when it is executed at the cloud side, and use t_i^e to denote the execution latency at the edge end device, respectively. Assuming the network bandwidth is B, we use $t_i^t = d_i/B$ to denote the transmission latency between v_i and its successor layers (if they are executed at different sides), where d_i denotes the size of output data of v_i.

The DNN model partition is to find a partition of V into two disjoint sets V_e and V_c, such that layers corresponding to vertices in V_e are executed at edge end devices and layers corresponding to vertices in V_c are executed at the cloud side, and the output data of nodes in V_e are transmitted from the end device to the cloud under the network bandwidth B. The execution latency at the edge side is $T_e = \sum_{v_i \in V_e} t_i^e$, the execution latency at the cloud side is $T_c = \sum_{v_j \in V_c} t_j^c$, and the transmission latency is $T_t = \sum_{(i,j) \in E_c} t_i^t$ where E_c denotes the set of edges connecting V_c and V_e. The total inference latency is thus $\mathcal{T} = T_e + T_t + T_c$, and we want to find an optimal partition of $\mathbf{M}$ that minimizes $\mathcal{T}$.

In the following, we first construct the latency graph $\widehat{\mathbf{G}}$ from $\mathbf{G}$, and then formulate the optimal model partition as a min-cut problem on $\widehat{\mathbf{G}}$.

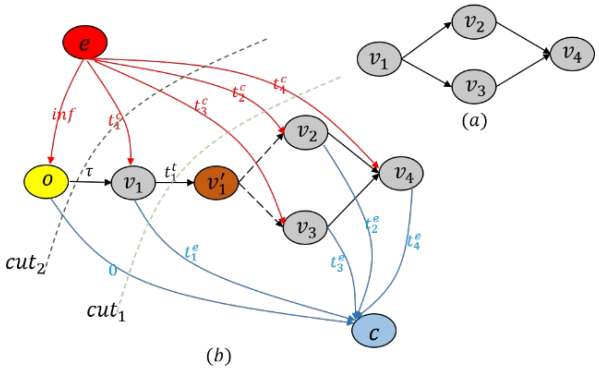


Fig. 5. Constructing the latency graph $\widehat{\mathbf{G}}$ from $\mathbf{G}$ representing the DNN model.

3.2.1 Latency Graph Construction. For simplicity in presentation, we use a simple model containing only four layers, whose corresponding DAG is shown in Figure 5(a), to demonstrate how to construct the corresponding latency graph (Figure 5(b)).

Given the graph $\mathbf{G} =< V, E >$ representing a DNN model $\mathbf{M}$, the corresponding latency graph $\widehat{\mathbf{G}}$ is constructed as follows.

Step 1: Add two vertices $\mathbf{e}$ and $\mathbf{c}$ to $\mathbf{G}$, which represent the starting point and the ending point of the model, respectively. For example, the vertices marked in red and blue in Figure 5(b).

Step 2: Add cloud latency to the obtained graph. For each vertex $v_i \in \mathbf{G}$, we and one edge $< \mathbf{e}, v_i >$ and assign its weight as t_i^c, e.g., v_i's layer latency when it is executed at the cloud side. For example, the red edges in Figure 5(b).

Step 3: Add edge latency to the obtained graph. For each vertex $v_i \in \mathbf{G}$, we and one edge $< v_i, \mathbf{c} >$ and assign its weight as t_i^e, e.g., v_i's layer latency when it is executed at the edge end device. For example, the blue edges in Figure 5(b).

Step 4: Add transmission latency between different layers. For each edge $< v_i, v_j > \in E$, we set the weight of the edge as $t_i^t = d_i/B$, where d_i is the size of v_i's output data and B is the network bandwidth.
In this step, we should pay special attention to those vertices with out-degree larger than 1, e.g., v_1 in Figure 5(b). Note that although v_1 has two out-edges, if we partition the model after v_1, the output data of v_1 need to be transmitted only once rather than two times. Thus, for each vertex $v_i \in \mathbf{G}$ with out-degree larger than 1, we perform the following operations: 1) remove all the edges starting from v_i, 2) add a new vertex v_i', 3) add a new edge $< v_i, v_i' >$ and set its weight as t_i^t, and 4) for each v_j to which there previously exists one edge $< v_i, v_j >$, add a new edge $< v_i', v_j >$ and set its weight as $+\infty$.

Step 5: Add practical constraints. In the last step, we add one vertex $\mathbf{o}$ and three edges $< \mathbf{e}, \mathbf{o} >$, $< \mathbf{o}, \mathbf{c} >$, and $< \mathbf{o}, v_1 >$, where v_1 is the vertex corresponding the first layer in $\mathbf{M}$. We set the weight of the three edges as $+\infty$, 0, and a model dependent value τ, whose value should satisfy $\tau > \sum_{v_i \in V}(t_j^e - t_j^c) + t_1^t$.

After the five steps, we construct a new graph $\widehat{\mathbf{G}}$ from $\mathbf{G}$. For example, the graph in Figure 5(b) is the latency graph constructed from Figure 5(a).

3.2.2 The Min-Cut Problem. After the latency graph is constructed, we can formulate the optimal partition of a DNN model $\mathbf{M}$ as a min-cut problem on $\widehat{\mathbf{G}}$:

Assume that we find a minimum edge cut C in the latency graph $\widehat{\mathbf{G}}$, which divides the vertices into two disjoint sets V_c and V_e. Assume that $\mathbf{e} \in V_e$ and $\mathbf{c} \in V_c$. Then we execute layers corresponding to vertices in V_e at the edge end device and execute layers corresponding to vertices in V_c at the cloud side. This model partition achieves the minimum total inference latency.

The proof of the above statement can be found in [9]. However, the DADS approach proposed in [9] constructs the latency graph through only Step 1 to Step 4. This makes it fail to find a meaningful solution in some settings (e.g., when $t_i^c \leq t_i^e$ for all layers). Actually, because the cloud is usually much more powerful than edge end devices, in practice it always takes less time to execute a layer on the cloud than on the end device, i.e., we always have $t_i^c \leq t_i^e$. The following proof shows that without Step 5, the obtained solution will always be a trivial one that puts all layers on the cloud.

Theorem 1. *Assume that $t_i^c \leq t_i^e$ for all layers in the model. If we remove $\mathbf{o}$ and the three edges connecting to it from the latency graph $\widehat{\mathbf{G}}$, then the min-cut on the resulting graph will be between $\mathbf{e}$ and all vertices in $\mathbf{G}$. In other words, in this case, the optimal partition will be executing all layers on the cloud.*

Proof. Assume that C be the minimum cut of $\widehat{G}$ after removing $\mathbf{o}$ and the three edges, and assume that C' is the cut that assigns all vertices in G to the cloud. For simplicity in presentation, we assume that $V_e = \{v_1, v_2 ... v_i\}$

and $V_c = \{v_{i+1}, v_{i+2}...v_n\}$, and the cut C contains k edges $\{e_1, \ldots, e_k\}$. Then for C the cutting cost is

$$c = \sum_{j=1}^{i} t_j^e + \sum_{j=i+1}^{n} t_j^c + \sum_{j=1}^{k} t_{s(e_j)}^t, \tag{1}$$

where $s(e_j)$ is the starting vertex of e_j. Similarly, the cutting cost for C' is

$$c' = \sum_{j=1}^{n} t_j^c. \tag{2}$$

Then we have

$$c - c' = \sum_{j=1}^{i} (t_j^e - t_j^c) + \sum_{j=1}^{k} t_{s(e_j)}^t > 0, \tag{3}$$

which contradicts with the assumption that C is the minimum cut. □

The reason for this problem is that if we put all layers on the cloud, we need to transmit the original input (e.g., the image) to the cloud, but previous works like [9] overlooked this problem when constructing the latency graph. In Step 5 in Section 3.2.1, we solve this problem by explicitly introducing the external transmission cost τ. If all the layers are deployed in the cloud, the cost of the cut C' will be $c' = \sum_{j=1}^{n} t_j^c + \tau$. Let $\Delta c = c - c'$. Then we have $\Delta c = \sum_{j=1}^{i}(t_j^e - t_j^c) + \sum_{j=1}^{k} t_{s(e_j)}^t - \tau$. If $\tau >= \sum_{j=1}^{i}(t_j^e - t_j^c) + \sum_{j=1}^{k} t_{s(e_j)}^t$ such that $\Delta c \leq 0$, then C will be the min-cut. Otherwise, if $\tau < \sum_{j=1}^{i}(t_j^e - t_j^c) + \sum_{j=1}^{k} t_{s(e_j)}^t$ such that $\Delta c > 0$, C' will be the min-cut, in which case all the layers will be deployed to the cloud. In both cases, the min-cut guarantees the optimal model partition.

We use the graph in Figure 5(b) to show that our latency graph construction method can always find the proper partition of the model. When $\tau > t_1^e - t_1^c + t_1^t$, the cost of *cut1* if smaller than the cost of *cut2*, and the model will be partitioned after v_1, in which case $\{v_1\}$ will be executed at the end device and $\{v_2, v_3, v_4\}$ will be executed at the cloud side. When $\tau < t_1^e - t_1^c + t_1^t$, the cost of *cut2* is smaller than the cost of *cut1*. In this case, all the four layers will be executed on the cloud.

3.3 Accurate Per Layer Latency Measurement

It is necessary to know the per layer latency of a DNN model on the cloud/end devices to construct the latency graph $\widehat{G}$ and calculate the min-cut. As we have discussed in Section 2.2, existing works [9, 12] do not consider the cross-layer optimization [6] that are widely used in popular machine learning frameworks such as Caffe [11], PyTorch [20], and TensorFlow [1], and thus cannot obtain accurate per layer latency measurement. In this section, we design a new method to measure per layer latency of a given model.

Our latency measurement method is as follows. For the i-th layer L_i, we use $T^e(1, i)$ and $T^c(1, i)$ to denote the total latency of executing all the i layers (from the first layer to the i-th layer) on the cloud and on the end device, respectively. Then we calculate the per layer latency of L_i at the cloud as $t_i^c = T^c(1, i) - \max_{j \in J} T^c(1, j)$, where J contains all precursor layers of L_i. In other words, the latency of L_i is calculated as the difference between the total latency of L_1 to L_i and the maximum total latency of L_1 to all of L_i's precursor layers. Similarly, the per layer latency of L_i at the edge end device is calculated as $t_i^e = T^e(1, i) - \max_{j \in J} T^e(1, j)$. Note that for different input the accumulative latency of a series of layers might be slightly different, we measure the latency on a large number of inputs and take the average value to construct the latency graph for each given DNN model.

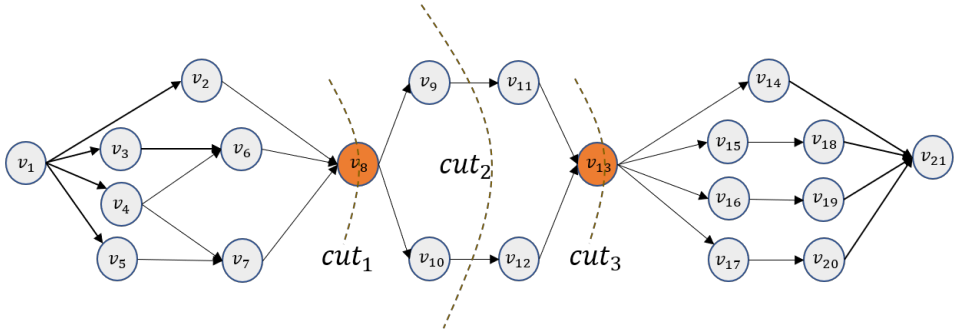


Fig. 6. Both cut_1 and cut_3 are cut at the cut vertex, and cut_2 is cut at the subgraph between the two cut vertices v_8 and v_{13}. The cut cost of cut_2 may be smaller than cut_1 and cut_3.

4 THE QDMP APPROACH

4.1 Design Guideline

After we construct the latency graph $\widehat{\mathbf{G}}$ for a DNN model, we can find the minimum cut of $\widehat{\mathbf{G}}$ obtain the optimal model partition accordingly. However, the time complexity of the min-cut algorithm is $O((n+m)n^2)$, where n and m represent the number of vertices and number of edges in $\widehat{\mathbf{G}}$. This might result in unacceptable large decision time to support real-time responses to network dynamics, as our experimental results show in Section 2.1. In this section, we propose a two-stage quick deep model partitioning approach that finds the min-cut of $\widehat{\mathbf{G}}$ with much lower average time complexity.

We first prove that the min-cut of $\widehat{\mathbf{G}}$ must be between some special vertices of $\widehat{\mathbf{G}}$ in Section 4.2, then describe our two-stage QDMP approach in Section 4.3, and finally analyze the time complexity of our algorithm in Section 4.4.

4.2 The Feature of the Min-Cut

We first define the cut vertex of a directed graph G as follows.

Definition 1 (Cut vertex). *Given a directed graph $G =< V, E >$, a vertex $v \in V$ is called a cut vertex of G if and only if the removal of v will increase the number of connected components in G.*

As an example, v_9 in Figure 4(b) is a cut vertex. A DNN model usually has more than one cut vertices. For example, the DAG corresponding to the GoogleNet has 38 cut vertices. The removal of a cut vertex will split the graph into at least two disjoint parts.

We find that the min-cut of the latency graph $\widehat{\mathbf{G}}$ must be between some two consecutive cut vertices of $\widehat{\mathbf{G}}$. The proof is as follows.

Theorem 2. *Given a DAG $\mathbf{G}$, its min-cut of $\mathbf{G}$ must be between two consecutive cut vertices of $\mathbf{G}$.*

Proof. Denote the min-cut of $\mathbf{G}$ as E_{cut}. Denote the vertex corresponding to the first layer as u and the vertex corresponding to the final output layer as v. There are two cases to consider:

(1) There are no cut vertices between u and v. In this case, Theorem 2 holds.
(2) There are at least one cut vertices between u and v. Without loss of generality, we consider the cut vertex that is closest to u and denote it as s. Note when selecting s we require there exists at least one cut edge in E_{cut} between u and s. If all edges in E_{cut} are between u and s, the case reduces to the first case and Theorem 2 holds. Otherwise, there must be some edges in E_{cut} that are between v and s. Denote by E'_{cut} the

set of cut edges between u and s and denote by E''_{cut} the set of cut edges between s and v. Neither E'_{cut} nor E''_{cut} is a cut of $\mathbf{G}$; otherwise, it contradicts with the assumption that E_{cut} is a min-cut. However, because E'_{cut} cannot split u and s, and E''_{cut} cannot split s and v, the graph is not partitioned, which contradicts the assumption that E_{cut} is a min-cut.

According to Theorem 2, we can first find all the cut vertices of $\mathbf{G}$ and find the min-cut by iterating over consecutive cut vertices. Because the subgraph between two consecutive vertices is much smaller than the original graph, this two-stage approach can significantly reduce time-complexity. For example, v_8 and v_{13} are two cut vertices in the graph shown in Figure 6. According to Theorem 2, the min-cut must be in the subgraphs between (v_1, v_8), (v_8, v_{13}), or (v_{13}, v_{21}).

4.3 The Two-Stage Approach

4.3.1 Get Cut Vertex Set V_{cut}. To calculate the cut vertex set of $\mathbf{G}$, we first get the undirected graph $\bar{G}$ corresponding to $\mathbf{G}$ and use the following method to calculate V_{cut}.

Theorem 3. *In the DFS tree of an undirected connected graph $\bar{G}$, the non-root vertex u is the cut vertex of $\bar{G}$ if and only if u has a child node v such that v and its descendants have no reverse edges connected back to the ancestors of u. If u is the root node and the number of u's children is greater than 1, then u is a cut vertex.*

Proof. Please refer to [26] for the proof.

According to Theorem 3, we can use the Tarjan algorithm [25] presented in Algorithm 1 to find the cut set of $\bar{G}$, namely V_{cut}. In the next subsection, we will generate a subgraph of $\mathbf{G}$ between two consecutive cut vertices and find the min-cut on the subgraph.

4.3.2 Get the Min-Cut. After obtaining the cut vertex set, we iterate all consecutive cut vertex pairs and find the min-cut on the subgraph between them. The min-cut with the minimum value among all the min-cuts on the subgraphs is the min-cut on the original graph $\mathbf{G}$. The process is as following.

Assume that there are k cut vertices in the graph, and denote them as $V_{cut} = \{cv_1, \ldots, cv_k\}$. Denote by $delay_{min}$ the minimal cost of $\mathbf{G}$, and initialize $delay_{min} = \infty$. Denote by V^e and V^c the set of vertices divided into the cloud and the end devices, respectively. V^e and V^c are initialized as $\emptyset$. The process to find the best cut set of $\mathbf{G}$ is as follows.

First, construct the subgraph between v_1 and cv_1 by using the method presented in Section 3.2.1. Note that τ is set as D_{input}/B, where D_{input} is the size of the input of the model and B is the network bandwidth. Denote the generated subgraph by $G^0_{sub} =< V^0_{sub}, E^0_{sub} >$. Find the min-cut on G^0_{sub} and denote the cutting cost as c^0. If $c^0 + T^c(\Pi(cv_1), v_n) < delay_{min}$[2], update $delay_{min} = c^0 + T^c(\Pi(cv_1), v_n)$. $V^e = \{v_1\} \bigcup V^e_0$ and $V^c = \{v_{\Pi(cv_1)}, \ldots, v_n\} \bigcup V^c_0$, where V^c_0 and V^e_0 are the layers partitioned into the cloud and the edge end device in G^0_{sub}, respectively.

Second, for each $1 \leq j < k$, construct the subgraph between cv_j and cv_{j+1}. Note the value of τ is set as D^j_{input}/B, where D^j_{input} is the total size of cv_j's input and B is the network bandwidth. Similarly, denote the generated subgraph by $G^j_{sub} =< V^j_{sub}, E^j_{sub} >$ and find the min-cut on it. Denote the cutting cost as c^j. If $c_j + T^e(1, \Pi(cv_j)) + T^c(\Pi(cv_{j+1}), n) < delay_{min}$, update $delay_{min} = c_j + T^e(1, \Pi(cv_j)) + T^c(\Pi(cv_{j+1}), n)$, $V^e = \{v_1, \ldots, v_{\Pi(cv_j)}\} \bigcup V^e_j$ and $V^c = \{v_{\Pi(cv_{j+1})}, \ldots, v_n\} \bigcup V^c_j$, where V^c_j and V^e_j are the layers partitioned into the cloud and the edge end device in G^j_{sub}, respectively.

Finally, construct the subgraph between cv_k and v_n, and update the minimal cost $delay_{min}$, the cloud side vertices V^c, the edge side vertices V^e similarly.

[2] $\Pi(cv_i)$ represents the index of cv_i in $\mathbf{G}$.

Algorithm 1 The Tarjan Algorithm.

Input: $\bar{G}$
Output: V_{cut}

$V_{cut} \leftarrow \{\}$;
$dfs\_clock \leftarrow 0$;
DFS($v_1, -1$); ▷ *Call DFS from* v_1
for $v \in V$ **do**
  if $iscut[v] == true$ **then**
    add v *to* V_{cut};
  end if
end for
return V_{cut};
function DFS($u, parent$)
  $child \leftarrow 0$;
  $dfs\_clock \leftarrow dfs\_clock + 1$;
  $low_u \leftarrow dfs\_clock$;
  $pre[u] \leftarrow dfs\_clock$;
  for $v \in adjacent\ node\ of\ u$ **do**
    if $pre[v] == 0$ **then**
      $child \leftarrow child + 1$;
      $low_v \leftarrow$ DFS(v, u);
      $low_u \leftarrow min(low_u, low_v)$;
      if $low_v > pre[u]$ **then**
        $iscut[u] \leftarrow true$;
      end if
    else if $pre[v] < pre[u]$ *and* $v! = parent$ **then**
      $low_u \leftarrow min(low_u, pre[v])$;
    end if
  end for
  if $parent == -1$ *and* $child == 1$ **then**
    $iscut[u] \leftarrow false$;
  end if
end function

After iterating all the cut vertices, we can find the best cut on the original graph **G**. The overall description of the QDMP algorithm is given in Algorithm 2.

4.4 Time Complexity Analysis

The QDMP algorithm consists of two stages. In the first stage, the Tarjan algorithm is used to find the cut vertex set, whose time complexity is $O(n+m)$ where n and m are the number of vertices and edges in **G**, respectively. In the second stage, we need to iterate over $k+1$ subgraphs. For each subgraph, it takes $O((n_j + m_j)n_j^2)$ to find the min-cut, where n_j and m_j are the number of vertices and number of edges in the j-th subgraph ($1 \leq j \leq k+1$), respectively. Let $\widehat{\boldsymbol{n}} = max(n_1, n_2, n_3, \ldots, n_{k+1})$ and $\widehat{\boldsymbol{m}} = max(m_1, m_2, m_3, \ldots, m_{k+1})$. Then the total time complexity in the second stage is $O((k+1)(\widehat{\boldsymbol{n}} + \widehat{\boldsymbol{m}})\widehat{\boldsymbol{n}}^2)$. Combining the two stages, the overall time complexity of QDMP is $O((k+1)(\widehat{\boldsymbol{n}} + \widehat{\boldsymbol{m}})\widehat{\boldsymbol{n}}^2)$.

Algorithm 2 The QDMP Algorithm.

Input: $\mathbf{G}$, $\{t_1^t, t_2^t, ...t_n^t\}$
Output: V^e, V^c

$delay_{min} \leftarrow \infty$;
$V^e, V^c \leftarrow \emptyset$;
$\tau \leftarrow D_{input}/B$;
$V_{cut} \leftarrow Tarjan(\mathbf{G})$;
$G_{sub}^0 \leftarrow graph\_construct(\mathbf{G}, v_1, cv_1, \tau)$;//Section 3.2.1
$V_0^e, V_0^e, c^0 \leftarrow mincut(G_{sub}^0)$;
if $c^0 + T^c(\Pi(cv_1), n) < delay_{min}$ **then**
    $delay_{min} \leftarrow c^0 + T^c(\Pi(cv_1), n)$;
    $V^e = \{v_1\} \bigcup V_0^e$
    $V^c = \{v_{\Pi(cv_1)}, \ldots, v_n\} \bigcup V_0^c$
end if
$k \leftarrow length(V_{cut})$;
for $j \leftarrow 1; j < k; j \leftarrow j + 1$ **do**
    $\tau \leftarrow D_{input}^j/B$;
    $G_{sub}^j \leftarrow graph\_construct(\mathbf{G}, cv_j, cv_{j+1}, \tau)$;
    $V_j^e, V_j^e, c^j \leftarrow mincut(G_{sub}^j)$;
    if $c^j + T^e(1, \Pi(cv_j)) + T^c(\Pi(cv_{j+1}), n) < delay_{min}$ **then**
        $delay_{min} \leftarrow c^j + T^e(1, \Pi(cv_j)) + T^c(\Pi(cv_{j+1}), n)$;
        $V^e = \{v_1, \ldots, v_{\Pi(cv_j)}\} \bigcup V_j^e$
        $V^c = \{v_{\Pi(cv_{j+1})}, \ldots, v_n\} \bigcup V_j^c$
    end if
end for
$\tau \leftarrow D_{input}^k/B$;
$G_{sub}^k \leftarrow graph\_construct(\mathbf{G}, cv_k, v_n, \tau)$;
$V_k^e, V_k^e, c^k \leftarrow mincut(G_{sub}^k)$;
if $c^k + T^e(1, \Pi(cv_k)) < delay_{min}$ **then**
    $delay_{min} \leftarrow c^k + T^e(1, \Pi(cv_k))$;
    $V^e = \{v_1, \ldots, v_{\Pi(cv_k)}\} \bigcup V_j^e$
    $V^c = \{v_n\} \bigcup V_k^c$
end if
return V^e, V^c

In practice, the vertices and edges are nearly equally distributed between cut vertices in popular DNN models, i.e., $n_j \approx \frac{n}{k+1}$ and $m_j \approx \frac{m}{k+1}$. In such scenarios, QDMP performs very well: its time complexity is close to $O(\kappa^2(n+m))$, where $\kappa \approx \frac{n}{k+1}$ represents the average number of vertices between two consecutive cut vertices. Because for most DNN models κ is a small number, QDMP can be executed in nearly linear time complexity, as the experimental results in Section 5 show. In contrast, the time complexity of the state-of-the-art DADS is $O((n+m)n^2)$, which increases significantly when the model becomes large.

Table 2. Inference latency (seconds) of typical DNN models with different partition approaches.

Model	3G					4G				
	C-O[1]	E-O[2]	NUERO[3]	DADS	QDMP	C-O	E-O	NUERO	DADS	QDMP
AlexNet	1.14	0.73	0.69	0.69	**0.43**	0.25	0.73	0.25	0.25	**0.22**
AlexNet-Parallel	1.12	0.36	0.36	0.36	**0.33**	0.23	0.36	0.36	**0.23**	**0.23**
DarkNet19	1.20	1.76	1.20	1.20	**0.92**	0.31	1.76	**0.31**	**0.31**	**0.31**
Tiny-YOLO	3.89	1.52	1.41	1.41	**0.91**	0.83	1.52	0.83	0.83	**0.65**
ResNet-18	1.23	0.80	0.80	0.70	**0.48**	0.34	0.80	0.34	0.34	**0.29**
GoogLeNet	1.32	1.17	1.17	1.17	**0.86**	0.43	1.17	**0.43**	**0.43**	**0.43**

[1]Cloud-Only. [2]Edge-Only. [3]Neurosurgeon.

5 PERFORMANCE EVALUATION

5.1 Experiment Setup

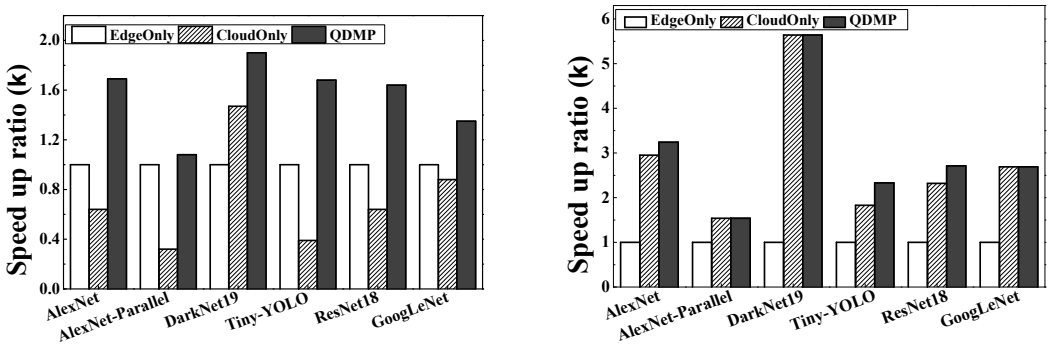
We implement the QDMP approach on real hardware. We use the Raspberry Pi 3B platform as the edge end device, which is equipped with a 4-core ARM Cortex-A53@1.2GHz processor and 1G RAM. For the cloud, we use a server equipped with an 8-core Intel core i7-9700k@3.60GHz processor and an NVIDIA RTX 2080i TX GPU. The BDD100K [36] self-driving dataset is used to evaluate the inference latency of different approaches. We sample every frame in the video of BDD100K, reshape each frame to a 224×224 RGB image in which each pixel is represented with 24 bits, and feed the image into different DNN models as the input.

The `iperf` command is used to obtain real-time bandwidth between the end device and the cloud. After the real-time bandwidth is obtained, QDMP is executed to decide which part of the given DNN model should be executed on the end device and which part should be executed at the cloud side. The input image is first inferred at the end device side to obtain the intermediate result, which is then transmitted to the cloud as the input of the layers at the cloud. The final inference result is then transmitted back to the edge end device.

We compare QDMP with two state-of-the-art solutions: DADS [9] and Neurosurgeon [12]. Neurosurgeon cannot partition DNN models with DAG topologies. To evaluate its performance on DAG models, we follow the method used in [9] which uses topological sorting to transform DAG models into chain-like models. Besides DADS and Neurosurgeon, we also consider two baseline approaches: The Edge-Only approach that executes all the layers at the edge end device and the Cloud-Only approach that transmits all the data to the cloud and executes all the layers at the cloud side. In the experiments, we consider six different DNN models, including three chain-like models and three DAG models. The three chain-like modes are AlexNet, Tiny-YOLO [21] and DarkNet19 [22]). The three DAG models are AlexNet-Parallel, ResNet-18, and GoogLeNet [24].

5.2 Performance Comparison

5.2.1 Inference Latency. We first compare the inference latency of different approaches under different network status. We use 3G and 4G as the default transmission technology, whose theoretical maximum uplink bandwidths are 1.1 Mbps and 5.85 Mbps, respectively. The inference latencies of different DNN models are listed in Table 2. It can be found that QDMP can always find the optimal model partition and achieve minimum latency. In contrast, DADS and Neurosurgeon cannot find proper model partition in most cases and result in longer inference latency than QDMP. In the 4G case, DADS can properly partition 3 models with optimized inference latency. We should point out the reason that DADS cannot properly partition a model is due to its inaccurate per layer latency measurement. If we use the per layer latency measured with the approach proposed in this paper, DADS should perform the same as QDMP in terms of inference latency.



(a) 3G (b) 4G

Fig. 7. Inference latency speed up ratio when compared to Cloud-Only and Edge-Only.

We further evaluate the speed up ratio of different partition approaches, with the latency of Edge-Only as the benchmark. The speed up ratio of the *ALG* approach is defined as

$$\kappa = \frac{L_{ALG}}{L_{EdgeOnly}}, \tag{4}$$

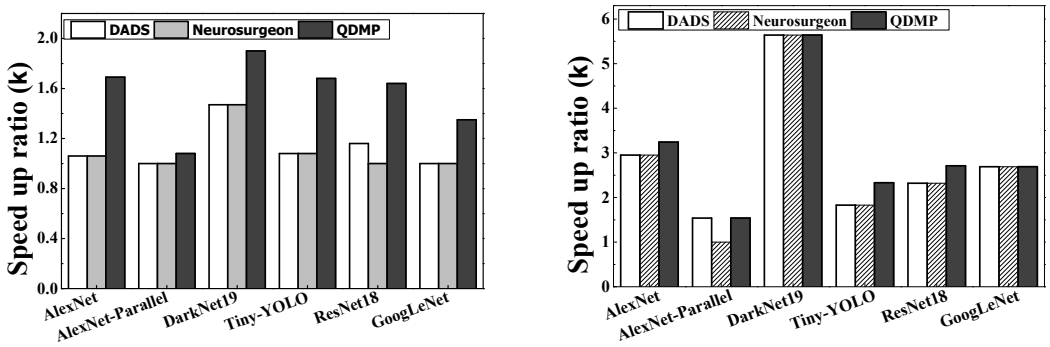
where L_{ALG} represents the inference latency of the *ALG* approach (e.g., DADS and QDMP), and $L_{EdgeOnly}$ represents the inference latency of Edge-Only. Note the speed up ratio of Edge-Only is 1. The results are plotted in Figure 7 and Figure 8.

Comparison with Edge-Only and Cloud-Only: Figure 7 plots the speed up ratio of QDMP and Cloud-Only for different models. It can be observed when the network bandwidth is low (e.g., 3G), Cloud-Only will result in longer inference latency (i.e., $\kappa < 1$) except for DarkNet19. In this case, the time spent in data transmission will be large and hence dominates the inference latency. In contrast, in the 4G case, Cloud-Only performs better than Edge-Only because computation latency takes most of the total inference latency. In both cases, QDMP achieves minimal inference latency for all the six models, with speed up ratio ranging from 1.08-1.90 for 3G case and 1.54-5.64 for 4G case.

Comparison with DADS and Neurosurgeon: Figure 8 plots the speed up ratio of DADS, Nuerosurgeon, and QDMP for different models. QDMP significantly outperforms Nuerosurgeon and DADS when the network bandwidth is low, as shown in Figure 8(a). Compared with DADS, QDMP reduces latency by 0.08-0.69 in the 3G case and by 0.08-0.41 in the 4G case. It can also be observed from Figure 8(b) that when the network bandwidth is high, both DADS and Neurosurgeon perform the same as or even worse than Cloud-Only. In these cases, QDMP can still find the proper partition to speed up the inference latency for three out of the six models.

5.2.2 Throughput. The throughput refers to the number of frames the DNN model can infer in one second. Here we adopt the same method as in DADS [9] to calculate the throughput as $\mathcal{T} = \frac{1}{\max\{T_e, T_t, T_C\}}$, where T_e, T_t, and T_c are the execution latency at the end device, the transmission latency, and the execution latency at the cloud side, respectively. The throughput of different models is listed in Table 3. Generally, the throughput increases when the bandwidth increases and QDMP outperforms all other methods in both cases.

Comparison with Edge-Only and Cloud-Only: Similar to latency, we normalize the throughput of different approaches by using the throughput of Edge-Only as the benchmark. The throughput ratios of QDMP and Cloud-Only are plotted in Figure 9. In both cases, QDMP significantly improves throughput. Compared to Edge-Only,



(a) 3G (b) 4G

Fig. 8. Inference latency speed up ratio when compared to DADS and Neurosurgeon.

Table 3. Throughput (#frames per second)of typical DNN models with different partition approaches.

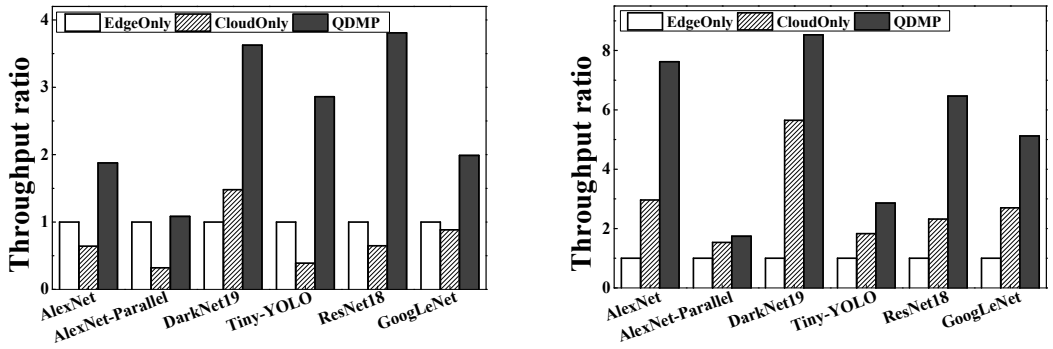
Model	3G					4G				
	C-O	E-O	NUERO	DADS	QDMP	C-O	E-O	NUERO	DADS	QDMP
AlexNet	0.88	1.37	1.54	1.54	**2.57**	4.06	1.37	4.86	4.86	**10.45**
AlexNet-P	0.89	2.79	2.79	2.79	**3.01**	4.28	2.79	2.79	**4.86**	**4.86**
DarkNet19	0.83	0.57	0.91	0.91	**2.03**	3.22	0.57	**4.86**	**4.86**	**4.86**
Tiny-YOLO	0.26	0.66	0.71	0.71	**1.89**	1.21	0.66	1.41	1.41	**1.89**
ResNet-18	0.81	1.25	2.74	1.25	**4.75**	2.91	1.25	4.86	4.86	**8.08**
GoogLeNet	0.76	0.86	0.86	0.86	**1.71**	2.32	0.86	**4.41**	**4.41**	**4.41**

QDMP improves throughput by up to 3.81x in 3G case and up to 8.52x in 4G case. Compared with Cloud-Only, QDMP improves throughput by up to 7.33x in 3G case and up to 2.79x in 4G case, respectively.

Comparison with Neurosurgeon and DADS: Compared with Neurosurgeon, QDMP increases throughput by up to 3.81x in 3G case and up to 2.15x in 4G case. Compared with DADS, QDMP increases throughput by up to 2.65x in 3G case and up to 2.15x in 4G case. QDMP significantly outperforms both Neurosurgeon and DADS when the bandwidth is low. Even when the bandwidth is high (e.g., in the 4G case), QDMP can effectively increase throughput for DAG models.

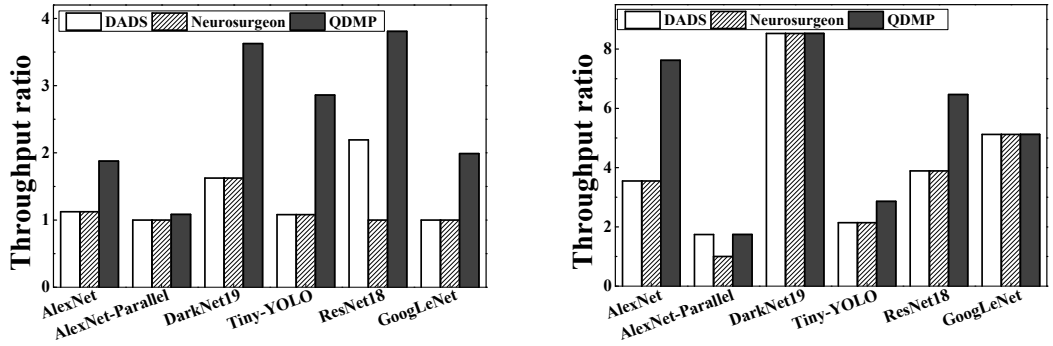
5.2.3 Decision Time. The time complexity of QDMP is smaller than that of DADS and it is much faster than DADS in practice. We evaluate the decision time of QDMP and DADS on six DNN models with different number of layers. Three types of end devices are used in the evaluation: Raspberry Pi 3B, Raspberry Pi 4B, and Intel i3-3240. The experimental results are shown in Table 4.

It can be observed from Table 4 that the decision time of QDMP is fairly stable when the number of layers increases. For GoogLeNet that has 166 layers, the decision time is only 1.68 seconds, less than 2 times of the decision time for AlexNet that contains only 24 layers. In contrast, the decision time of DADS on GoogLeNet is more than 68 seconds, nearly 500 times of the decision time for AlexNet. On the Raspberry Pi 3B platform, QDMP reduces decision time by up to 40.8x for large models such as GoogLeNet. Moreover, the speed up ratio becomes more significant when the number of layers in the model increase. It can also be observed that when the model is small, DADS might outperform QDMP on some low-end devices like Raspberry Pi 3B. The reason is that QDMP



(a) 3G (b) 4G

Fig. 9. Throughput comparison with Cloud-Only and Edge-Only.



(a) 3G (b) 4G

Fig. 10. Throughput comparison with Neurosurgeon and DADS.

Table 4. Decision time (seconds) of different approaches on different hardware platforms.

Model	#Layer	Raspberry Pi 3B		Raspberry Pi 4B		Intel i3-3240	
		DADS	QDMP	DADS	QDMP	DADS	QDMP
AlexNet	24	0.14	1.09	0.05	0.05	0.01	0.02
AlexNet-Parallel	28	0.16	1.12	0.06	0.06	0.02	0.02
DarkNet19	27	0.17	1.09	0.05	0.05	0.02	0.02
Tiny-YOLO	16	0.10	1.04	0.03	0.03	0.01	0.01
ResNet-18	111	18.14	1.45	4.88	0.18	1.88	0.05
GoogLeNet	166	68.42	1.68	18.18	0.27	6.82	0.09

needs to construct the latency graph for k times where k is the number of cut vertices. When the model is small, the time spent in graph construction takes a large part in the decision time. In contrast, when the model is large, the time spent in graph construction can be neglected as the major time is spent on solving the min-cut problem.

Table 5. Maximum uplink bandwidth (Mbps) of different wireless transmission technologies.

CAT1	3G	4G	Wi-Fi
0.13	1.1	5.85	18.88

Table 6. Network traffic (Mb) to process one frame under different network status.

	CAT1	3G	4G	Wi-Fi
AlexNet	0.05	0.05	0.53	1.15
AlexNet-Parallel	0.02	0.02	1.15	1.15
DarkNet19	0.01	0.38	1.15	1.15
Tiny-YOLO	0.00	0.19	1.15	1.15
ResNet-18	0.10	0.19	0.38	1.15
GoogLeNet	0.00	0.33	1.15	1.15

We also compare the decision time of QDMP and DADS on two types of powerful end devices, namely a Raspberry Pi 4B platform and a personal computer equipped with an Intel i3-3240 processor. On the two devices, QDMP performs the same as DADS for small models but significantly outperforms DADS for large modes such as GoogLeNet and ResNet-18. On the Raspberry Pi 4B platform, QDMP is 27.1x faster for ResNet-18 and 67.3x faster for GoogLeNet. Even on the powerful personal computer, DADS still needs nearly 7 seconds to partition GoogLeNet. As a comparison, QDMP takes only 0.09 seconds to partition GoogLeNet on the same device, 75.8x faster than DADS. Moreover, as analyzed in Section 4.4, the decision time of QDMP increases linearly with the size of the latency graph of a model, while the decision time of DADS increases much faster.

5.2.4 Network Traffic Reduction. QDMP can save network traffic by transmitting only intermediate layer output data, whose size is usually much smaller than the size of raw data. The volumes of data that are transmitted in different models to process one frame are listed in Table 6. Four typical wireless transmission scenarios are considered: CAT1, 3G, 4G, and Wi-Fi. Table 5 lists the maximum uplink bandwidth for these transmission technologies.

In our experiment, the size of each frame is $224 \times 224 \times 24 \approx 1.15$ Mb. Thus, in Cloud-Only it needs to transmit 1.15 Mb data to the cloud side to process one frame. With QDMP, the end device can adaptively partition the model to reduce the network traffic according to the bandwidth. When the network bandwidth is low (e.g., CAT1 and 3G), QDMP tends to execute all the layers on the end device or transmit only a few data. As shown in Table 6, QDMP saves network traffic by 98% in CAT1 case and 83% in 3G case. When the network bandwidth is relatively high, QDMP tends to execute part of or all the layers on the cloud side. In 4G case, QDMP saves network traffic by up to 67% and 20% on average. In the Wi-Fi case in which the bandwidth is high, QDMP tends to put all layers on the cloud side.

5.3 Impact of Bandwidth Variation

In this section, we use the ResNet-18 model as an example to evaluate how the fine-grained bandwidth variation affects the model partition. We increase the bandwidth from 0 Mbps to 15 Mbps stepped by 0.5 Mbps, and measure the latency speed up ratio (κ) of QDMP against Edge-Only and Cloud-Only. The results are plotted in Figure 11.

It can be observed from Figure 11 when network bandwidth is low, executing the model on the end device is superior to execute it on the cloud side, because the transmission time dominates the total inference latency in such cases. When the bandwidth is higher than 2 Mbps, the latency of cloud-Only is smaller than that of Edge-Only. QDMP is most suitable when the network bandwidth is between 0 Mbps and 12 Mbps, in which case

it can effectively speed up the inference by finding the optimal model partition. When the network bandwidth is higher than 12 Mbps, the optimal solution would be transmitting the data to the cloud and executing the model at the cloud side.

It should be noticed that the turning point of the bandwidth depends on the model and the size of the input. In our experiment, the size of the input is not very large, approximately 150 KB per frame. If the size of the input data is large (e.g., 1 MB), QDMP is able to reduce latency even when the bandwidth is higher than 15 Mbps.

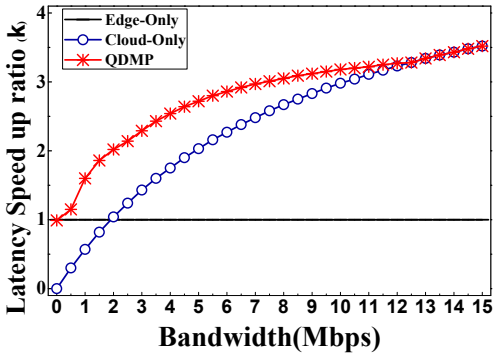


Fig. 11. ResNet-18 inference latency speed up ratio for different bandwidth.

5.4 Cutting Edge Selection

Under the four different bandwidths listed in Table 5, we analyze the edge cuts of different DNN models. Figure 12 shows the cutting edge of the three chain topological DNN models at different bandwidths. In the Cat1 network and the 3G network, AlexNet is partitioned between the first pooled layer and the first fully connected layer. In the 4G network, AlexNet is cut between the first convolution kernel layer and the second pooling layer. For DarkNet19, it is partitioned between the penultimate convolutional layer and the first to last convolutional layer in the Cat1 network, and partitioned between the 8th convolutional layer and the 4th fully connected layer in the 3G network. In the 4G network, the optimal solution for DarkNet19 is to deploy the entire model at the cloud side. Tiny-YOLO should be deployed at the edge end device in the Cat1 network. In both 3G and 4G cases, Tiny-YOLO should be partitioned between the 5th convolutional layer and the 5th pooled layer. All the chain-topological models should be deployed to the cloud side in the Wi-Fi network.

Figure 13 shows the cutting edges of three DAG-topological DNN models at different bandwidths. In the Cat1 and the 3G networks, AlexNet-Parallel is partitioned between the bottom two layers and the third last layer. In the 4G network, the cut edge set selected by AlexNet-Parallel is between the second parallel convolution layer and the second parallel pooling layer. In the Cat1 network, the cut edge set selected by ResNet-18 is between the last residual block layer and the fully connected layer. In the 3G network, the cut edge set selected by ResNet-18 is in the middle of the subgraph constructed by the third residual block layer. In the 4G network, the cut edge set selected by ResNet-18 is between the first BN layer and the first pooled layer. In the Cat1 network, GoogLeNet chooses to deploy the model to the edge. In the 3G network, GoogLeNet should be partitioned in the middle of the fourth Inception block layer. In the 4G network, GoogLeNet chooses to deploy the model to the cloud. All the DAG models choose to deploy the model to the cloud in the Wi-Fi network.

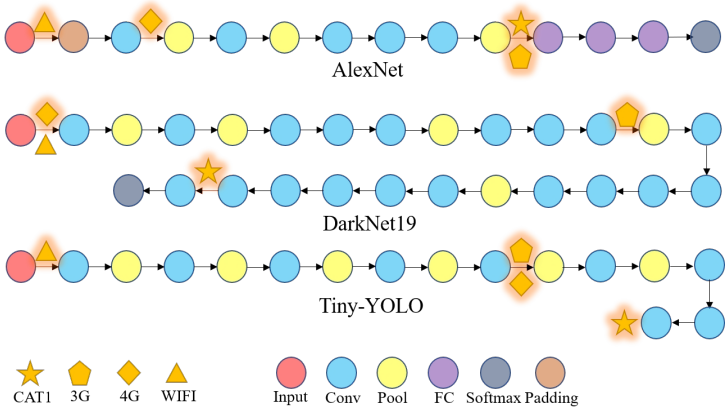


Fig. 12. Partition of chain-topological models in different networks listed in Table 5.

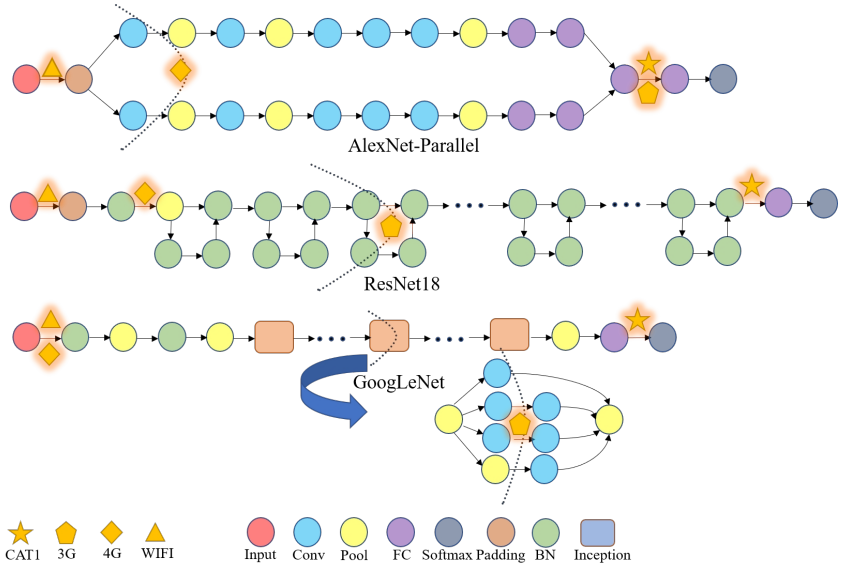


Fig. 13. Partition of DAG-topological models in different networks listed in Table 5.

6 LIMITATIONS AND FUTURE RESEARCH DIRECTIONS

6.1 Limitations

One major limitation of current deep learning model partition approaches, including QDMP, DADS [9], and Neurosurgeon [12], is that they cannot partition non-DAG Models. Neurosurgeon can handle only linear models with chain topology. QDMP and DADS make a step forward and can partition deep learning models that can be represented as a DAG. However, there are some deep learning models that cannot be represented as a DAG, such as models built on recurrent neural networks (RNN) and long short-term memory (LSTM) [31]. In these models, data might be transmitted from a rear layer to a layer in front of it, forming a loop between layers. For example,

the recent developed Bert model [4] and XLNet model [34] cannot be represented as DAG and thus cannot be partitioned by QDMP and DADS. In the future, we need to design new approaches to partition such models.

6.2 Future Research Directions

The work on cooperative deep inference can open several potential research directions. First, cooperative deep learning model execution can boost AI-enabled IoT applications in edge-cloud computing environments. Current IoT applications face a great challenge in handling the huge amount of data generated by a large number of IoT devices. Due to the severe constraints on the end device's resources, it is impractical to directly run heavyweight deep learning models on IoT end devices, which greatly limits the utilization of deep learning models in IoT applications. Through optimal model partition and distributed model execution, IoT applications can now exploit the power of deep learning models to be smarter. Second, in the future, it might need to train a deep model locally on the end devices, in which case the method proposed in this paper would be helpful. Currently, the models are usually trained on the cloud platform and then deployed to end devices or the edge server. In the future, we can link end devices, edge servers, and the cloud platform to jointly train and tune the deep model to reduce the volume of data to be transmitted between end devices and the cloud. Finally, cooperative inference is helpful to enhance data privacy in smart IoT applications that employ deep learning models to perform task inference. Recent studies [2] have shown that deep learning models might memorize some features of the data and thus cause privacy leakage of the data. With cooperative deep inference, this problem can be mitigated because only intermediate results rather than the raw data are processed by the model at the cloud side.

7 RELATED WORK

7.1 Deep Model Surgery

One strategy to reduce the execution latency of a deep learning model is to offload a part or all of the computation of the model to a powerful cloud platform. Neurosurgeon [12] initiated deep model partitioning to exploit resources on both end devices and the cloud platform to improve the execution efficiency of a deep model. It divides a DNN into two parts and distributes the execution between a mobile device and the cloud platform to accelerate the execution of the model. However, Neurosurgeon has two limitations. First, it can partition only chain-like models but cannot properly handle models with a DAG structure. Second, it uses a linear regression method to predict the running time of each layer in the model, which is inaccurate. There are some improvements to Neurosurgeon. In [13], one JPEG encoder is used to compress the intermediate result to further reduce the volume of data to be transmitted between the end device and the cloud. In DDNN [28], the authors study how to reduce the inference latency in a distributed neural network that spans end devices, edge servers, and a cloud platform. However, DDNN is designed for BranchyNet [27] that has a special tree topology and thus is difficult to be extended to other types of deep learning models. In DeepX [14], the authors design a resource scheduling algorithm on the edge side to cut the DNN into multiple parts and allocate them to different processors for processing according to the required resources.

Edgent [16] combines DNN partitioning and an early exit strategy [28] to predict the running time of each layer and find the best cutting point in a dynamically changing network. JointDNN [6] performs collaborative calculations for DNNs between mobile devices and the cloud during the inference and training phases. It proposes a continuous layer measurement method and uses the shortest path algorithm to find the optimal partition of the model subject to minimum latency and energy consumption on the end device. However, it is not suitable for DNN models with DAG topology. DADS [9] aims to partition DNN models with a DAG topology. However, the model used in DADS is flawed, making it fail to obtain proper partition in practical settings, e.g., when the per layer latency on the end device is shorter than that on the cloud. Moreover, the time complexity of DADS is very high that it takes a long time to find the optimal partition for a large model, fails to provide real-time response to

network status changes. Considering that the partition decision is made on resource-constraint end devices, the decision time should be short in order to adapt to the dynamic changes of network status quickly.

7.2 Model Weight Reduction

Another strategy to improve the execution efficiency of deep models on resource-constraint devices is to design lightweight models or compress existing models. For lightweight models or compressed models, the number of model parameters is reduced and thus they require less memory than the original models. However, such models usually require dense computations such as 1×1 convolution operation, resulting in longer execution time. Furthermore, the accuracy of the models might also be impacted after model compression. MobileNet [8] uses a depth-wise separable convolution operation based on SqueezeNet [10] to compress the model. By using the depth-wise convolution operation, MobileNet reduces the number of model parameters and increases the speed of the operation. In ShuffleNet [38], group convolution based on depth-wise separable convolution is used to group input features and implement efficient feature group fusion through channel shuffle operations. Model compression removes the relatively small weight in the DNN by cutting filters and neurons, thereby reduces the memory required to store the model and improves the speed of the operation. In [17, 19], the authors propose a filter-level pruning method based on the similarity between filters to compress the model. In [35], the authors propose a neuron-level pruning method to explore the positive-negative correlation between features and prune unimportant neuronal nodes. ECC [32, 33] takes the energy consumption as the constraint objective to guide model pruning operation. It constructs a constrained optimization process to prune the model and compress the size of the DNN. After pruning, the model volume is greatly reduced and the speed of reasoning is also improved, making it suitable to be deployed at the edge end devices with limited resources.

8 CONCLUSION

In this paper, we propose QDMP, a time-efficient approach to deep learning model partition in edge computing environments. QDMP can obtain optimal model partition with second-level decision time even for large models containing hundreds of layers. The merit of QDMP is its novel two-stage design, in which cut vertices of the model are first obtained and then the optimal partition is found by iteratively calculating min-cuts on subgraphs between adjacent cut-vertices. We propose a correct latency construction method and an accurate per layer latency measurement method, with which the optimal model partition can be properly found for a large variety of deep learning models. Testing on real hardware demonstrates up to 66.3x speedup in decision time when compared with state-of-the-art model partition solutions. One limitation of QDMP is that it cannot handle deep models that contain loops between layers. Time-efficient model partition for such models, e.g., deep learning models built on RNN or LSTM, will be our future work.

ACKNOWLEDGMENTS

This work is partially supported by the National Defense Science and Technology Innovation Special Zone Project of China, the National Natural Science Foundation of China (Grant Nos. 61772559, 61872310, 61672543 and 61972145) and the General Research Fund of the Research Grants Council of Hong Kong (PolyU 152221/19E).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] Mohammad Al-Rubaie and J Morris Chang. 2019. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy* 17, 2 (2019), 49–58.

[3] Mark Billinghurst, Adrian Clark, Gun Lee, et al. 2015. A survey of augmented reality. *Foundations and Trends in Human–Computer Interaction* 8, 2-3 (2015), 73–272.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.

[5] Swarnava Dey, Arijit Mukherjee, Arpan Pal, and P Balamuralidhar. 2018. Partitioning of cnn models for execution on fog devices. In *Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing*. 19–24.

[6] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2019. JointDNN: an efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing* pp (2019), 1–12.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[8] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[9] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge. In *Proceedings of 38th Conference on Computer Communications (Infocom)*. 1423–1431.

[10] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[12] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. 45, 1 (2017), 615–629.

[13] Jong Hwan Ko, Taesik Na, Mohammad Faisal Amir, and Saibal Mukhopadhyay. 2018. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *Proceedings of the 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. 1–6.

[14] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE Press, 23.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[16] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2019. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications* (2019).

[17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *Proceedindgs of the 5th International Conference on Learning Representations(ICLR)*.

[18] Xuan Liu, Quan Yang, Juan Luo, Bo Ding, and Shigeng Zhang. 2019. An Energy-Aware Offloading Framework for Edge-Augmented Mobile RFID Systems. *IEEE Internet of Things Journal* 6, 3 (2019), 3994–4004.

[19] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*. 8024–8035.

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 779–788.

[22] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 7263–7271.

[23] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, Yoshua Bengio and Yann LeCun (Eds.).

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 1–9.

[25] Robert Endre Tarjan. 1974. A note on finding the bridges of a graph. *Information Processing Letter* 2 (1974), 160–161.

[26] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.

[27] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *Proceedings of the 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2464–2469.

[28] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 328–339.

[29] Vutha Va, Takayuki Shimizu, Gaurav Bansal, Robert W Heath Jr, et al. 2016. Millimeter wave vehicular communications: A survey. *Foundations and Trends in Networking* 10, 1 (2016), 1–113.

[30] Xiaofei Wang, Yiwen Han, Chenyang Wang, Qiyang Zhao, Xu Chen, and Min Chen. 2019. In-edge AI: Intelligentizing mobile edge computing, caching and communication by federated learning. *IEEE Network* 33, 5 (2019), 156–165.

[31] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. 2015. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In *Proceedings of Advances in neural information processing systems (NIPS)*. 802–810.

[32] Haichuan Yang, Yuhao Zhu, and Ji Liu. 2019. Ecc: Platform-independent energy-constrained deep neural network compression via a bilinear regression model. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 11206–11215.

[33] Haichuan Yang, Yuhao Zhu, and Ji Liu. 2019. Energy-Constrained Compression for Deep Neural Networks via Weighted Sparse Projection and Layer Input Masking. In *Proceedings of International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=BylBr3C9K7

[34] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*. 5754–5764.

[35] Jaehong Yoon and Sung Ju Hwang. 2017. Combined group and exclusive sparsity for deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 3958–3966.

[36] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. 2018. BDD100K: A Diverse Driving Video Database with Scalable Annotation Tooling. *CoRR* abs/1805.04687 (2018). arXiv:1805.04687 http://arxiv.org/abs/1805.04687

[37] Shigeng Zhang, Yinggang Li, Bo Liu, Shupo Fu, and Xuan Liu. 2019. Enabling Adaptive Intelligence in Cloud-Augmented Multiple Robots Systems. In *Proceedings of the 13th IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE.

[38] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6848–6856.