2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00053

LoADPart: Load-Aware Dynamic Partition of Deep Neural Networks for Edge Offloading

Hongzhou Liu*, Wenli Zheng*, Li Li†, Minyi Guo*
* School of Electronic Information and Electric Engineering, Shanghai Jiao Tong University, Shanghai, China
† State Key Laboratory of IoTSC, University of Macau, Macau, China
deanlhz@sjtu.edu.cn, zheng-wl@cs.sjtu.edu.cn, LLiLi@um.edu.mo, guo-my@cs.sjtu.edu.cn

***Abstract*—The emerging edge computing technique provides support for the computation tasks that are delay-sensitive and compute-intensive, such as deep neural network inference, by offloading them from a user-end device to an edge server for fast execution. The increasing offloaded tasks on an edge server are gradually facing the contention of both the network and computation resources. The existing offloading approaches often partition the deep neural network at a place where the amount of data transmission is small to save network resource, but rarely consider the problem caused by computation resource shortage on the edge server. In this paper, we design LoADPart, a deep neural network offloading system. LoADPart can dynamically and jointly analyze both the available network bandwidth and the computation load of the edge server, and make proper decisions of deep neural network partition with a light-weighted algorithm, to minimize the end-to-end inference latency. We implement LoADPart for MindSpore, a deep learning framework supporting edge AI, and compare it with state-of-the-art solutions in the experiments on 6 deep neural networks. The results show that under the variation of server computation load, LoADPart can reduce the end-to-end latency by 14.2% on average and up to 32.3% in some specific cases.**

***Index Terms*—Deep Neural Networks, Inference, Computation Offloading**

I. Introduction

Along with the prosperity of Deep Learning (DL), Deep Neural Networks (DNNs) are widely employed in diverse scenarios, such as image classification, speech recognition, recommendation, and machine translation. Those applications can demand high timeliness for satisfactory user experience. Furthermore, the emerging edge computing technologies enable responsive, cost-efficient and secure machine intelligence at the network edge. These advantages facilitate both real-time and intensive computation tasks like DNN inference, which is the kernel of machine intelligence, at the edge. However, the impressive amount of computation of DNNs on the edge devices with restricted computation resources becomes a vital challenge.

To address the problem of lacking computation resources, a major category of approaches is to curtail the computation demand generated by *local inference*, i.e., executing the inference task on the user-end device. The DNNs that are designed specially for devices with restricted resources, such as SqueezeNet[1] and EfficientNet[2], aim at achieving relatively high accuracy while reducing inference time and energy consumption. Meanwhile, DNN compression techniques are proposed to further reduce computation, for instance, parameter pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation[3].

However, as the DNNs are growing more complicated for higher accuracy, sometimes employing devices with much stronger computation ability than mobile and embedded devices to meet the computation demand becomes a must. Hence another category of approaches is *full offloading*, i.e., uploading the DNN input features to edge servers, which have stronger computation capabilities than the user-end devices, and then perform inference on them. However, the end-to-end latency of an inference task is likely to be affected by the unstable network connection or the limitation of bandwidth between the user-end devices and the edge server, which harms the timeliness.

Therefore, *partial offloading* [4], [5] is presented with each DNN partitioned and distributed across the edge server and user-end device to enable collaborative inference. In this way, an optimized partition point can minimize the amount of communication data, and the computation capability of the edge server can be also utilized. Nonetheless, the increasing number of IoT and mobile devices is expected to make edge servers busier than ever before. The existing DNN offloading approaches do not address the contention of computation resources, which can bottleneck the collaborative inference on busy edge servers. To address this problem, the DNN offloading system should dynamically and jointly analyze both the available network bandwidth and the server computation load when optimizing the partition point. In addition, as the architectures of DNN become more sophisticated, the DNN partition algorithms which only support chain architectures cannot cope with sophisticated direct acyclic graph (DAG) architectures, while the DNN partition algorithms that can deal with sophisticated DAG architectures incur high time complexities, and thus are not suitable for dynamic partition. To adapt to the variation of the available network bandwidth and the server computation load, a light-weighted partition algorithm that can support DNNs with sophisticated DAG architectures is required.

In this work, we propose LoADPart, a novel DNN offloading system to address the aforementioned problems. We consider the variation of both the available network bandwidth and the server computation load to foster load-aware dynamic

2575-8411/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDCS54860.2022.00053

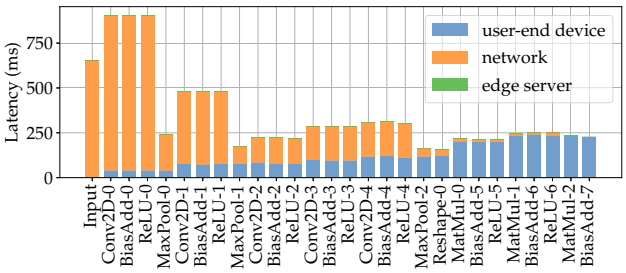

Fig. 1: The end-to-end latency when AlexNet is partitioned at different points. The network bandwidth is 8Mbps for both upload and download.

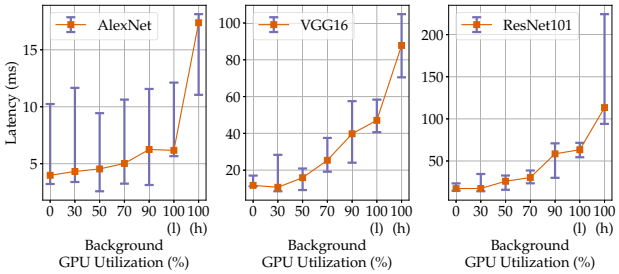

Fig. 2: The end-to-end latency of the inference of AlexNet, VGG16, and ResNet101 under different computation load levels. The square on each error bar represents the average.

DNN partitioning and edge offloading, thereby achieving a resource-contention-aware collaborative inference. We first develop the prediction models for the execution time of DNN layers, which are used by the DNN partition algorithm. Then, we formulate the optimization problem of where to partition a DNN for the minimized total latency and propose a light-weighted but efficient algorithm to solve the problem. Finally, we implement LoADPart for MindSpore[6], an emerging DL framework, to extend its ability of efficient execution across the edge server and user-end devices. We compare LoADPart with state-of-the-art approaches on 6 DNNs. The results show that under the variation of server computation load, LoADPart can reduce the end-to-end latency by 14.2% on average and up to 32.3% in some specific cases.

II. Motivation

We take AlexNet, which has a chain architecture[7] and is thus easy to analyze, to illustrate the benefits and challenges of DNN partition and offloading in the scenario of edge computing. We partition AlexNet after each of its layers (each time after a different layer), including the input layer, and offload the computation of all the layers after the partition point to the edge server. We measure the end-to-end latency for each partition scheme and present the result in Figure 1. Each bar is divided into the computation latency on the user-end device, the network transmission overhead, and the computation latency on the edge server (the last of the three is almost negligible when the server is not busy). The hardware specifications of the edge server and user-end device in this measurement are shown in Table IV. When the partition point is right after MaxPool-2, the end-to-end latency is reduced up to 4× compared with full offloading, i.e., executing the whole AlexNet on the edge server (the leftmost bar), and is reduced by 30% compared with local inference, i.e., executing the whole AlexNet on the user-end device (the rightmost bar). The result reveals that we can carefully choose a partition point where (1) the output tensor size is smaller than the initial input tensor size of a DNN, and (2) the execution time to perform the inference computation before the chosen partition point on the user-end device is not too long.

More importantly, we observe how the computation load of the edge server (specified in Table IV) affects the DNN inference latency of AlexNet, VGG16[8], and ResNet101[9]. The input feature maps of these DNNs are all in the shape of $3 \times 224 \times 224$ with a batch size of 1. We generate six levels of background computation load (which can be caused by other tasks, e.g., inference tasks offloaded from other user-end devices) on the GPU of the edge server by running 7 processes, and each of the process execute AlexNet on the input of the shape $1 \times 3 \times 224 \times 224$ periodically. We adjust the period of the execution to acquire different GPU utilizations from 30% to 100%(l), where "l" stands for "low". For the background GPU utilization 100%(h), where "h" stands for "high", we execute ResNet152 on the input of the shape $1 \times 3 \times 224 \times 224$ every 1μs concurrently in 7 processes for extremely high background computation load. Though the GPU utilizations are the same for the case 100%(l) and the case 100%(h), the queueing time of each GPU kernel of the background tasks differs in the two cases, which can also affect the foreground DNN inference latency significantly.

We sample the end-to-end latencies of AlexNet, VGG16, and ResNet101 every 15ms for 1000 times under different background computation load levels. The result is illustrated in Figure 2. It indicates that when the computation load level of the edge server is below 50%, although sometimes the end-to-end latency fluctuates, its average is slightly hurt. GPU is underutilized under such levels of background computation load, and therefore, the inference tasks can usually be scheduled and executed immediately on the GPU. However, when the computation load becomes heavier (above 90%), the GPU is fully utilized and the inference tasks are queued up to await being scheduled and executed. Thus, the average end-to-end latency increases. Meanwhile, the end-to-end latencies all fluctuate significantly under such a heavy load due to the uncertainty of how the aforementioned causes affect the execution of the tasks. To sum up, the computation load of the edge server can hurt the timeliness of DNN inference on it. It motivates us to consider the computation load of the edge server seriously when dynamically partitioning and offloading the DNN to mitigate the influence of such background load on the timeliness of DNN inference.

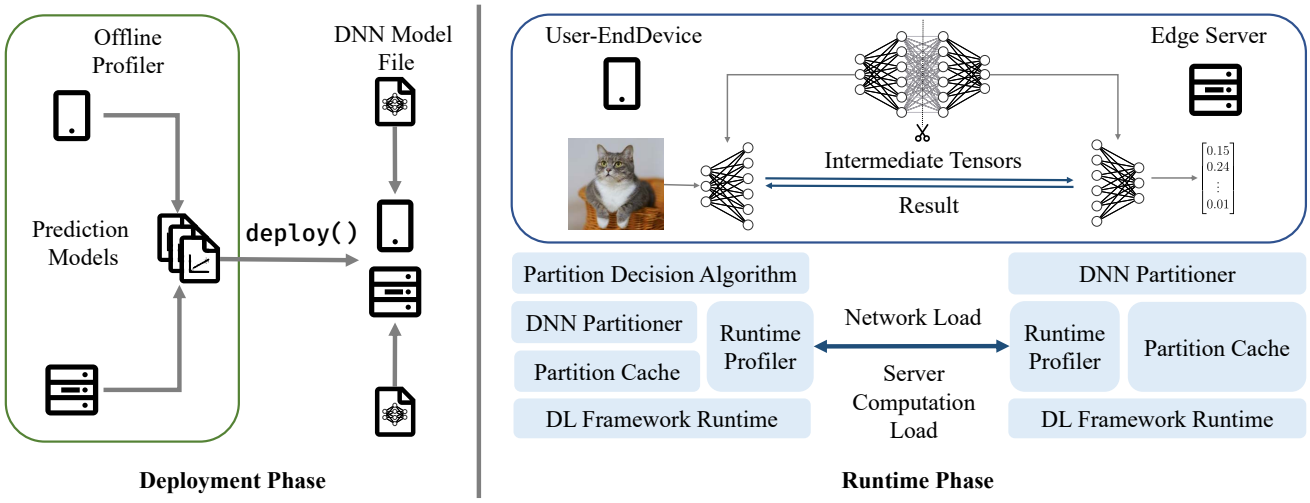

Fig. 3: The system diagram of LoADPart.

III. System Design

In order to determine the optimal partial offloading scheme, we design the LoADPart system with two major components, an offline profiler that trains the models predicting the inference time of various DNN layers, and an algorithm that decides the optimal partition point for offloading, based on the models trained offline and the load-related metrics collected in the runtime. In this section, we first overview how LoADPart works in general, and then explain the two major components in detail. At last, we briefly present the method of implementing LoADPart for MindSpore, a DL framework supporting edge AI.

A. Overview of LoADPart

LoADPart is designed with an offline profiler that collects the execution times together with the layer configurations of various DNN layers to train the inference time prediction models for those layers. The trained prediction models are stored on both the user-end device and the edge server. When LoADPart works online, as shown in Figure 3, it first loads those models on both the two sides together with the DNN model file. Then during the runtime phase, the runtime profiler collects the available network bandwidth information and the server computation load information periodically. When the user requests an inference, the partition decision algorithm finds the optimal partition point based on the available network bandwidth and the server computation load in the current period, as well as the inference time prediction models trained offline. To avoid unnecessary data transmissions incurred by acquiring the partition points from the edge server, the partition decision algorithm runs on the user-end device. Then, the DNN is partitioned into two parts according to the optimized partition point. The first part is executed in the DL framework runtime on the user-end device, and the produced intermediate tensors are transferred to the edge server together with the partition point. Later, the edge server also partitions the DNN, executes the second part, and transfers back the result tensors to the user-end device.

In addition, to reduce the overhead of partitioning DNNs, we design a cache to store the partitioned DNNs, such that the overhead can be amortized and thus is negligible compared to the DNN inference time (only takes up about 1% of the inference time when amortized over 100 offloading requests). The cache takes the partition point of a DNN as the key and stores the corresponding partitioned computation graph and other auxiliary data structures for inference. Each time the partition decision algorithm returns a partition point on the user-end device, the DNN partitioner searches the cache with the partition point. If the cache hits, the partitioner does not need to perform DNN partition and prepare the DL framework runtime for inference. The situation is similar on the edge server. When it receives the partition point from the user-end device, its DNN partitioner also checks its cache to decide whether the DNN partition is needed.

B. Development of Inference Time Prediction Models

To obtain the optimal partition point, the execution time of each layer in the to-be-deployed DNNs must be acquired in advance on both the user-end device and edge server. Instead of executing the DNNs online case-by-case to acquire such execution time, which can easily introduce random measurement errors, we develop inference time prediction models for the typical types of DNN layers to provide layer-grained inference time prediction for diverse DNNs. The inference time of a DNN partition can then be calculated by adding up the predicted inference times of all its layers.

How to develop the inference time prediction models is highly impacted by where to conduction the prediction (on the edge server or user-end device). Conducting the prediction on the edge server would provide more computation resources for complicated models. For example, some existing approaches leverage specific DNNs to predict the inference time of typical

TABLE I: The FLOPs of 8 typical kinds of computation nodes.

Computation Node	$FLOPs$
Conv	$NC_{in}H_{out}W_{out}K_HK_WC_{out}$
DWConv	$NC_{in}H_{out}W_{out}K_HK_W$
Matmul	$NC_{in}C_{out}$
Pooling	$NC_{out}H_{out}W_{out}K_HK_W$
BiasAdd	$\prod_{i=0}^{n} S_i$
Element-wise	
BatchNorm	
Activation	

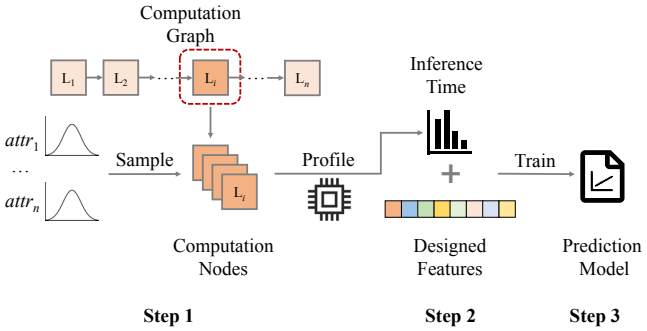

Fig. 4: The three-step process of developing inference time prediction model of a specific kind of computation node.

DNN layers (we discuss them in Section VI) and can reach high prediction accuracies. However, it also causes additional data transmissions, since the inference time prediction is periodically triggered by the DNN partition algorithm. If the inference time prediction runs on the edge server, the user-end device needs to request for either the prediction results or the partition decision every period, depending on whether the partition algorithm runs on the user-end device or edge server. Conducting the prediction as well as the partition algorithm on the user-end device can avoid such additional data transmissions, but lead to the necessity of less time-consuming and resource-consuming models due to the restricted computation capabilities. To make fast partition decisions on the user-end device, we select to trade off a little prediction accuracy for saving time and resource. Thereby, we leverage Linear Regression (LR) to model the computation time of various DNN layers by carefully designing the input features of LR models.

The development of the inference time prediction model of a specific kind of DNN layer is a three-step process, which is illustrated in Figure 4. In the first step, we profile the execution time of typical kinds of DNN layers. As a DNN layer can be mapped to multiple computation nodes in the computation graph, which is a fine-grained representation of the corresponding DNN in DL frameworks, we model the inference time for 8 typical kinds of computation nodes separately. We denote the inference time prediction models of computation nodes as M_{user} and M_{edge} for the user-end device and edge server respectively. In the second step, we design the feature vectors of the LR model based on our knowledge about the

TABLE II: The input features of the inference time prediction models for 8 typical kinds of computation nodes.

Computation Node	Edge Server	User-end Device
Conv	$FLOPs$, s_f, $H_{in}s_f$, $C_{out}s_f$	
DWConv	$FLOPs$, s_f, $padded\_size$	$FLOPs$, $NC_{out}s_f$
Matmul	$FLOPs$, NC_{in}, NC_{out}, $C_{in}C_{out}$	
Pooling	$FLOPs$, $NC_{in}H_{in}W_{in}$, $NC_{out}H_{out}W_{out}$, $H_{out}W_{out}$	
BiasAdd	$FLOPs$	
Element-wise		
BatchNorm		
Activation		

computation characteristics of the DNN layer. The inference time of a computation node is significantly impacted by the FLOPs (floating point operations). We calculate the FLOPs for the 8 typical kinds of computation nodes and present them in Table I. N is the batch size, and C, H, and W represent channel, height, and width of the input or output feature map respectively. K_H and K_W are the sizes of the filter of the convolutional and pooling computation nodes. S_i is the shape of the i-th axis of the input tensor and thus $\prod_{i=0}^{n} S_i$ is the total size of the input tensor. In addition to the FLOPs, we find the inference time of different layers/computation nodes can be also impacted by different other features. We discuss them respectively as follows.

a) Convolutional Layer: The convolutional layer is usually mapped to a convolutional computation node (Conv) and a bias-add computation node. It is not enough to take $FLOPs$ as the only input feature to satisfy the accuracy need. Thus, we list some other features related to the computation and memory access characteristics and leverage XGBoost[10] to score the feature importance. We select the features with high importance scores as the additional input features. We denote the size of a single filter as $s_f = C_{in} \times K_H \times K_W$ and the input features are presented in Table II.

We observe that except for the typical convolutional computation node, the depth-wise convolutional node also appears in some DNNs like Xception[11]. Therefore, we develop a model for depth-wise convolutional layers separately which is denoted as DWConv in Table II, where $padded\_size$ refers to the total size of the padded input feature map.

As for the bias-add computation node, because each element in the input tensor is added once by the corresponding element in the bias tensor, it is enough to take $FLOPs$ as the only input feature of both M_{user} and M_{edge}.

b) Fully-Connected Layer: The fully-connected layer consists of a matrix multiplication node (Matmul) and a bias-add node. Aside from the FLOPs, we consider the sizes of the input, weight, and output tensors as the input features for the matrix multiplication node, which are shown in Table II.

c) Pooling Layer: There are mainly two kinds of pooling layers, which are average pooling and max pooling. They share the same characteristics of computation. A pooling layer is mapped to a pooling computation node in the computation

TABLE III: The performance of the inference time prediction models of the edge server and user-end device.

Computation Node	Edge Server		User-End Device	
	RMSE (μs)	MAPE	RMSE (μs)	MAPE
Conv	401.81	16.71%	41325.68	40.09%
DWConv	11.95	41.58%	712.79	36.64%
Matmul	3.41	5.33%	420.71	8.54%
AvgPooling	6.90	13.56%	635.26	19.29%
MaxPooling	6.19	34.23%	2375.42	20.25%
BiasAdd	4.60	7.40%	690.55	4.80%
Elem-wise Add	1.47	6.37%	1232.25	4.82%
BatchNorm	24.34	10.97%	2023.16	9.36%
ReLU	4.52	12.59%	1451.52	17.67%

graph. We also utilize XGBoost for feature selection and all the input features are shown in Table II.

d) Activation, Batch Normalization, and Element-wise Add Layers: We develop the prediction models for ReLU, sigmoid, softmax and tanh activation layers as well as batch normalization layer and element-wise add layer. All these layers are each mapped to one computation node respectively in the computation graph. They are all element-wise operations, and thus we take the FLOPs as the only input feature of both M_{user} and M_{edge} for those computation nodes.

After the aforementioned two steps, the third step is to train the LR model by fitting the non-negative least squares (NNLS) [12] to keep all its regression coefficients positive and not fitting the intercept, to make sure when the input feature is a zero vector, the predicted inference time is zero. The training and testing data are collected during the first step. We investigate some common DNNs to decide the value ranges of attributes of different computation nodes. Then, for each kind of computation node, we sample uniformly in its corresponding ranges and profile the inference time of the computation nodes with the sampled attributes. We test the trained models with the testing data and report the performance of the models for both the edge server and the user-end device in Table III. We use the rooted mean squared error (RMSE) and the mean absolute percentage error (MAPE) as performance metrics. For element-wise and activation computation nodes, we report the performance of the element-wise add and the ReLU as examples respectively. The similar performance can be observed for developed models of other kinds of nodes belonging to these two categories. The performance of the developed models is sufficient to accurately determine the optimal partition point during the runtime.

C. Impact of Computation Load on Prediction Models

Since the above inference time prediction models for the edge server are developed under the background GPU utilization of 0%, we need to additionally consider the impact of computation load in online prediction. The previous work [4], [5] addresses this problem by involving a server load factor in the models for layer-grained inference time prediction, but does not give the details of how the factor is obtained and how the models involving the factor are developed. To assess the impact of server computation load on the inference time of a DNN layer, we take background GPU utilization as the quantitative indicator of the computation load and collect the execution times of different kinds of DNN layers under the utilization of 30%, 50%, 70%, 90%, and 100%. It turns out that the distributions of execution times are almost the same as the ones under 0% GPU utilization, except that the execution time of convolutional layers under some specific configurations are affected by the computation load. The reason is that the execution time of a single layer, in the most cases, is too short to use up a time slice of a time-multiplexed GPU (e.g., 2 ms). As a result, the tiny inference task is immediately scheduled and completed, and thus its execution time is not affected.

Nevertheless, the execution of a DNN partition consisting of multiple layers can use many time slices and be interrupted by the preemptive scheduler due to the resource contention with other tasks, and thus the inference time is lengthened (notice that the preemption does not happen when a certain layer is being executed, but rather before or after the layer is executed, due to the non-preemptive property of GPU kernels). Thus, we enable LoADPart to analyze the impact of the computation load on the predicted time of a certain DNN partition, rather than the impact on a single layer. Specifically, LoADPart monitors the actual execution time of the DNN partition on the edge server, records those in the most recent monitoring period, and calculates their average value. The ratio of this average value over the model-predicted execution time of the current DNN partition is taken as the influential factor k of the computation load. Then, LoADPart multiplies the model-predicted execution time of each potential DNN partition by this influential factor, to predict their execution times under the current computation load.

D. Design of the Partition Decision Algorithm

Our algorithm that makes the partition decision is designed to handle the computation graphs, which represent DNNs in DL frameworks. The computation graph is a DAG, whose nodes can be roughly classified into computation nodes and parameter nodes. We define the DAG formed by removing the parameter nodes in the computation graph as the backbone DAG of the computation graph, and denote it as $G(V, E)$. When partitioning a DNN, we actually partition its corresponding backbone DAG and then restore the parameter nodes to form a partition of the original computation graph.

As the DNNs are becoming more complex, the architectures of their backbone DAGs may become more challenging to making the optimal partition decision. For instance, there are Residual Blocks in ResNet and Inception blocks in Inception[13], which contain multiple branches. Solving optimization problems that find the optimal partition point on such DAGs usually costs $\mathcal{O}(n^3)$ time by a min-cut algorithm[14]. To make fast partition decisions on resource-constrained user-end devices, a light-weighted algorithm to find the optimal partition point is necessary. We deal with this challenge by firstly exploring whether the optimal partition point can be inside a block containing branches, and if not, the search

space can be significantly reduced. We generate all possible partition schemes for SqueezeNet, ResNet, Inception, and Xception, which are DNNs with DAG architectures. Then we investigate the transmission sizes of each scheme. We conclude that cutting the DAG inside any of the blocks incurs large transmission sizes and is hardly possible to generate the optimal partition. For example, the least transmission size when cutting InceptionV3 in its last Inception block is 1.25MB. It means that compared to the initial input size which is 1.02MB ($1 \times 3 \times 299 \times 299$), the transmission size can be even larger when cutting it in the earlier blocks.

Hence, we can find the optimal partition point by searching the topological order of the backbone DAG in linear time and partition the order into two parts. To make the formulation of the optimization problem concise, we add a virtual input node L_0 into the vertex set V and add an edge to connect it with the source node L_1 of G to form a new DAG $G'(V', E')$, where $V' = V \cup \{L_0\}$ and $E' = E \cup \{\langle L_0, L_1 \rangle\}$. We denote the topological order of such a DAG as $\{L_0, L_1, \cdots, L_n\}$, where $n = |V|$. It is clear that the partition of the topological order forms a cut $C(S, T)$ of the DAG G', where $L_0 \in S$ is the source node and $L_n \in T$ is the sink node. S and T are disjoint sets of the vertex set V. We also denote the transmission size of the intermediate tensor when the partition point is after L_i as s_i. Thus, s_0 is the input tensor size and s_n is the output tensor size. Then, with the inference time prediction models of the user-end device $f(\cdot)$ and the ones of the edge server $g(\cdot)$, we can make optimal partition decisions during the DNN offloading. Here $f(\cdot)$ depends on M_{user}, and $g(\cdot)$ depends on both M_{edge} and k, the influential factor of the server computation load. The optimization problem that minimizes the end-to-end inference latency is defined as follows.

$$\min_{p} \quad t_p = \begin{cases} \sum_{i=0}^{p} f(L_i) + \frac{s_p}{B_u} + & p \leq n-1 \\ \sum_{i=p+1}^{n} g(L_i, k) + \frac{s_n}{B_d} & \\ \sum_{i=0}^{n} f(L_i) & p = n \end{cases} \tag{1a}$$

$$\text{s.t.} \qquad 0 \leq p \leq n, \tag{1b}$$

$$k \geq 1, \tag{1c}$$

$$f(L_i), g(L_i, k), s_i \geq 0 \quad \forall i = 0, \cdots, n, \tag{1d}$$

$$B_u, B_d \geq 0 \tag{1e}$$

B_u and B_d represent the available upload and download bandwidths of the network between the user-end device and edge server respectively, and p is the partition point. Notice that $f(L_0) = g(L_0, k) = 0$ as L_0 is virtual, i.e., not executed physically, and $g(L_i, k)$ is a non-decreasing monotonic function with respect to k. When $p = n$, the partition point is at the end of the topological order, which indicates local inference; when $p = 0$, it indicates full offloading.

Now, we present our partition decision algorithm to solve Problem 1 with linear search in Algorithm 1. We utilize a prefix sum array and a suffix sum array to store $\sum_{i=0}^{p} f(L_i)$ and $\sum_{i=p+1}^{n} g(L_i, k)$ respectively to avoid redundant computations. Thus, the time complexity of the algorithm is $\mathcal{O}(n)$ and the space complexity is also $\mathcal{O}(n)$.

Algorithm 1: Partition Decision Algorithm

Input:
The number of the computation nodes: n
The topological order of the computation nodes V': $\{L_0, L_1, \cdots, L_n\}$
The transmission sizes: $\{s_0, s_1, \cdots, s_n\}$
The upload and download network bandwidths: B_u, B_d
The influential factor of the computation load of the edge server: k
The inference time prediction models for the user-end device and the edge server: $f(L_i), g(L_i, k)$
Output: The optimal partition point: p

Function `PartitionDecision`
  $prefix[0] \leftarrow 0$;
  $suffix[n+1] \leftarrow 0$;
  for $i \leftarrow 1$ **to** $n+1$ **do**
    $prefix[i] \leftarrow prefix[i-1] + f(L_{i-1})$;
    $suffix[n-i+1] \leftarrow suffix[n-i+2] + g(L_{n-i+1}, k)$;
  end
  $minVal \leftarrow +\infty$;
  for $i \leftarrow 1$ **to** $n+1$ **do**
    if $i = n+1$ **then**
      $curVal \leftarrow prefix[i]$;
    else
      $curVal \leftarrow prefix[i] + \frac{s_{i-1}}{B_u} + suffix[i] + \frac{s_n}{B_d}$;
    end
    if $curVal \leq minVal$ **then**
      $minVal \leftarrow curVal$;
      $p \leftarrow i - 1$;
    end
  end
  return p

IV. Implementation

We implement LoADPart by modifying MindSpore. First, we carefully tailor the partition decision algorithm (Algorithm 1) We have $f(L_i) = M_{user}(L_i)$ and $g(L_i, k) = k \times M_{edge}(L_i)$, where $M_{user}(L_i)$ and $M_{edge}(L_i)$ are the prediction models developed in Section III-B, and k is the influential factor of the server computation load defined in Section III-C. In this way, LoADPart can calculate the prefix and suffix sum arrays for only once at the beginning, and multiply the most recent value of the factor k on the suffix sum as $k \times \sum_{i=p+1}^{n} M_{edge}(L_i)$ when dynamically updating the partition decisions. For those computation nodes without developed inference time prediction models, we assign 0 to both $f(\cdot)$ and $g(\cdot)$, so they have no impact on the partition decision. Additionally, we ignore the network latency of downloading the result tensor from the edge server to the user-end device,

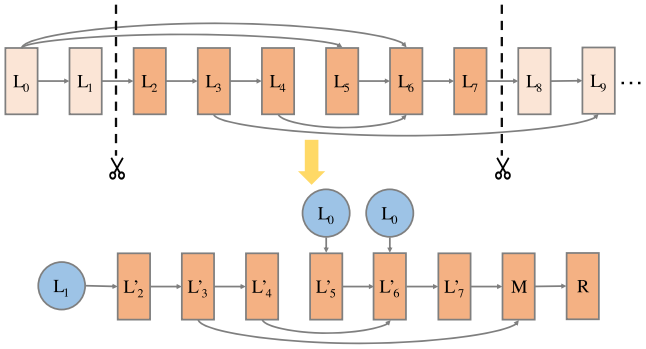

Fig. 5: An example of generating a computation graph from a segment of `CNodes`, which are illustrated as rectangles. "M" and "R" represent `MakeTuple` and `Return` nodes respectively. The `Parameters` are illustrated as circles and we only show newly added `Parameters`.

i.e., $\frac{s_n}{B_d}$. This is because such latency is much shorter than the upload latency and the inference latencies on both the user-end device and the edge server, due to the much smaller size of the output tensor of discriminative DNNs. Therefore, measuring the download bandwidth becomes insignificant.

As for partitioning a DNN, the actual data structure being partitioned is the corresponding computation graph, which is represented by MindIR in MindSpore. A computation graph contains mainly computation nodes and parameter nodes, called `CNode` and `Parameter` respectively. The topological order, as defined in Section III-D, is a list of `CNodes`. Given an arbitrary segment of the list, we first check the direct predecessors of all the `CNodes` and generate corresponding `Parameters` for the predecessors out of the segment, because their outputs are the inputs of the segment. Then, we find all the `CNodes` whose direct successors are out of the segment, because their outputs are the inputs of the next segment. We construct a `MakeTuple` computation node if the number of such `CNodes` is larger than 1 and link it to a `Return` computation node, which outputs the generated graph. We illustrate an example of generating a computation graph from a segment of `CNodes` in Figure 5. We apply this method to the segments of the topological order before and after p, and in this way, the computation graph gets partitioned into two parts on the user-end device and the edge server respectively.

The runtime profilers run on both the user-end device and edge server. On the user-end device, it is a thread running concurrently with the main thread that performs DNN offloading. The profiler thread measures the available upload bandwidth in two ways. The first way is periodically sending probe packages. It keeps a sliding window for upload bandwidth records and the window size can be defined by users. The size of the probe package is adjusted according to the historical data in the sliding window. The upload bandwidth is also tested passively by measuring the network latency of the DNN offloading in the main thread and added to the sliding window.

TABLE IV: Hardware specifications of the edge server and user-end device.

Hardware	Edge Server	User-End Device
System	Supermicro SYS-7049GP-TRT	Raspberry Pi 4 Model B
CPU	2× Intel Xeon Gold 6230R, 26C52T, 2.10GHz	ARM Cortex A72, 4C, 1.50GHz
Memory	4× 64GB DDR4 3200MHz	4GB LPDDR4 1600MHz
Hard Disk	2× 1T SSD + 2× 8T HDD	16GB microSD card
GPU	NVIDIA Tesla T4 16GB	N/A

On the edge server, the runtime profiler periodically calculates k, the influential factor of computation load, as introduced in Section III-C. When the runtime profiler on the user-end device asks for the computation load of the edge server, the profiler on the edge server side replies it with the most recent value of k. However, the runtime profiler on the user-end device can lose track of the computation load level of the edge server after the user-end device executes the whole DNN locally for a period of time. To address this issue, we run another thread to monitor the GPU utilization concurrently with the main thread that provides the offloading service on the edge server. Once the GPU utilization is under a threshold (e.g., 90%), the runtime profiler modifies the value of k, and thus the user-end can be notified that the GPU on the edge server has become underutilized when it asks for the computation load from the edge server.

V. Evaluation

In this section, we first present the experiment settings, and then evaluate LoADPart under the variation of network bandwidth and server computation load respectively.

A. Experiment Settings

The hardware specifications of the edge server and user-end device are defined in Table IV. They communicate with each other via a WiFi connection. The runtime profiler work with a period of 5 seconds, which can be shortened if the network bandwidth or server computation load fluctuate more frequently. The thread that monitors the GPU utilization on the edge server checks if the utilization is below 90% every 10 seconds, and updates the influential factor of server computation load when the user-end device temporarily conducts local inference.

We test the inference of AlexNet, VGG16, ResNet18, ResNet50, SqueezeNet, and Xception with LoADPart. AlexNet and VGG16 have chain architectures while the other 4 DNNs have DAG architectures. The input size is $1 \times 3 \times 227 \times 227$ for SqueezeNet and is $1 \times 3 \times 299 \times 299$ for Xception. For the rest of DNNs, the input sizes are all $1 \times 3 \times 224 \times 224$. In Section V-B, we vary the network bandwidth from 1Mbps to 64 Mbps to evaluate LoADPart under the variation of available network bandwidth. We fix the upload bandwidth as 8Mbps in Section V-C and evaluate LoADPart under varying server computation load.

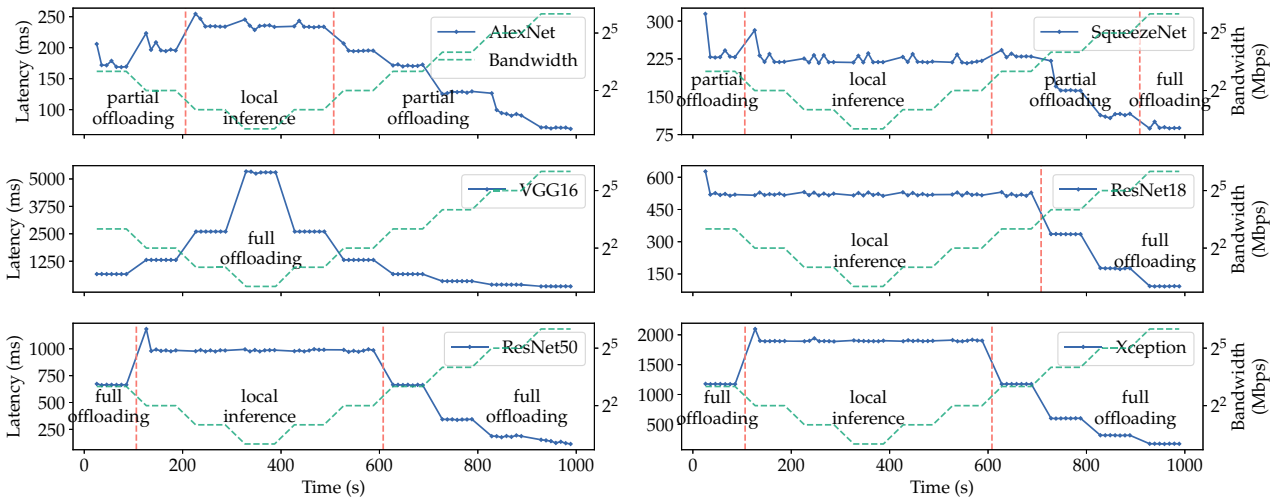

Fig. 6: The end-to-end latencies of the inference of AlexNet, SqueezeNet, VGG16, ResNet18, ResNet50, and Xception using LoADPart with the variation of the upload bandwidth (the x-axes are the same for all the subfigures).

B. Awareness of the Network Bandwidth

The evaluation results of LoADPart for the 6 DNNs under the variation of the network bandwidth are presented in Figure 6. The upload bandwidth starts from 8Mbps and decreases to 1Mbps, and then increases to 64Mbps. The result of AlexNet shows that when the upload bandwidth is relatively high, the algorithm decides to partition AlexNet at early points ($p = 4, 8$) and more leverage the computation power of the edge server. As the upload bandwidth starts to decrease, the decided partition point shifts towards the end of AlexNet ($p = 19$). When the upload bandwidth becomes scarce, e.g., 2Mbps, the algorithm chooses not to partition AlexNet but to conduct the inference locally instead ($p = 27$). As for SqueezeNet, the algorithm decides to partition it at $p = 39$ when the upload bandwidth is 8Mbps and at $p = 5$ when the bandwidth is 16Mbps or 32Mbps. It also decides to conduct inference locally when the upload bandwidth is 4Mbps and to offload the SqueezeNet fully to the edge server when the bandwidth is 64Mbps.

We also compare the end-to-end latencies under various network bandwidths among local inference, full offloading, and LoADPart for AlexNet and SqueezeNet in Figure 7 and Figure 8 respectively. When we use LoADPart, the speedup of the inference of AlexNet is $6.96\times$ on average and is up to $21.98\times$ compared with full offloading. The speedup compared to local inference is $1.75\times$ on average and is up to $3.37\times$. As for SqueezeNet, the speedup is $7.05\times$ on average and is up to $23.93\times$ compared to full offloading. The two speedup numbers are $1.41\times$ and $2.53\times$ respectively, compared to local inference. The results show the efficacy of our partition decision algorithm to find the optimal partition points and speed up the inference of AlexNet and SqueezeNet under various network bandwidths.

When it comes to VGG16, LoADPart decides to execute

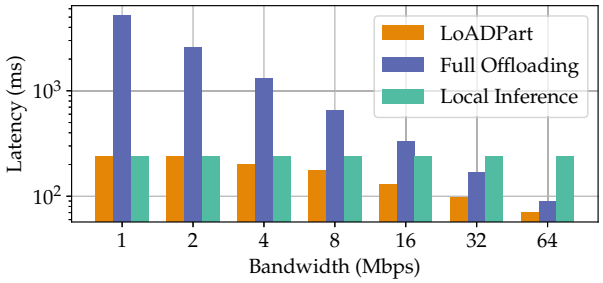

Fig. 7: The end-to-end latencies of the inference of AlexNet under different upload bandwidths.

the whole VGG16 on the edge server even when the upload bandwidth is extremely low (1Mbps). We investigate the actual inference time of each layer (or say computation node) of VGG16 and find our user-end device is so slow that executing any layer of VGG16 on it can increase the latency compared to full offloading. Take the earliest available partition point of VGG16 ("available" means the transmission size is smaller than the initial input size) $p = 23$ as an example. The inference time of all the layers before the partition point on the user-end device is 4875.7ms while transmitting the initial input tensor to the edge server under the upload bandwidth of 1Mbps costs 4593.8ms. In this case, partial offloading is not beneficial. We ascribe this limitation to the large gap of the computation capabilities between the user-end device and the edge server.

For ResNet18, ResNet50, and Xception, LoADPart decides to either conduct the inference locally or offload the whole DNN onto the edge server, which is also due to the limited computation capability of the user-end device. Nonetheless, it still adapts to the varying available network bandwidth

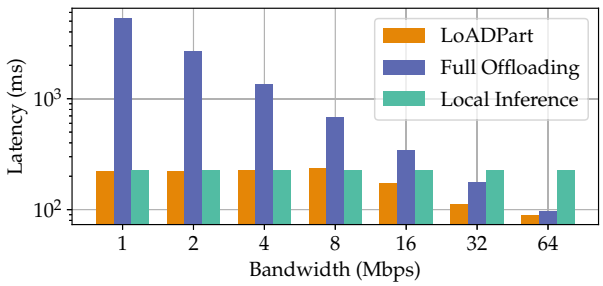

Fig. 8: The end-to-end latencies of the inference of SqueezeNet under different upload bandwidths.

successfully, by conducting inference locally under low bandwidths and offloading the DNNs fully onto the edge server under high bandwidths. We can also expect that when the computation capability of the user-end device gets improved (at least more powerful than the Raspberry Pi in our testbed), the end-to-end latency can be reduced by partial offloading.

To sum up, the evaluation results in this subsection show that LoADPart can adjust the partition point of the DNN to adapt to the variation of available network bandwidth and thus keep minimizing the end-to-end inference latency.

C. Awareness of the Computation Load of the Edge Server

In this section, we evaluate the end-to-end inference latencies of 6 DNNs under the variation of server computation load. We choose Neurosurgeon[4] as the baseline, which supports network bandwidth aware offloading but does not adjust the partition point when the server computation load varies. Following the results in Section II, we know the timeliness of offloading DNNs can be hurt largely when the GPU utilization is high. Thus, we generate the background GPU utilization from 0% to 100%(l) and then from 100%(l) to 100%(h). The evaluation results are presented in Figure 9.

For AlexNet, it shows that LoADPart shifts the partition point from $p = 8$ towards $p = 19$ when the server computation load becomes heavy. We can also observe that when the load is heavy, the end-to-end latency fluctuates a lot, which is consistent with our observation in Section II. Compared to the baseline, which keeps the partition point the same as LoADPart gives under the background GPU utilization of 0% ($p = 8$), LoADPart reduces the end-to-end latencies by 4.95% on average and up to 39.4%.

As for the result of SqueezeNet, we can observe the same trend that LoADPart shifts the partition point from $p = 39$ to $p = 99$ as the GPU utilization increases from 100%(l) to 100%(h). We can also observe that around 100s, there are protrusions of the latency, which are caused by the lag of acquiring the latest influential factor of computation load from the edge server. In addition, when the inference is conducted locally ($p = 99$) but the server computation load decreases, LoADPart successfully recognizes the change of the GPU utilization and thus can notify the user-end device to shift the partition point from $p = 99$ back to $p = 39$ (see the results around 220s). Compared to the baseline, which fixes the partition point at $p = 39$, LoADPart reduces the end-to-end latencies by 14.2% on average and up to 32.3%.

For VGG16 and Xception, LoADPart decides to offload them fully to the edge server, even when the server computation load is quite high (see the results between 150s and 220s). The latency of local inference of VGG16 is about 5200ms, which is larger than the maximum end-to-end latency of full offloading (around 1100ms) under heavy server computation load. In addition, the latency of local inference of Xception is 1800ms, while the maximum end-to-end latency of full offloading is 1760ms. As LoADPart aims at minimizing the end-to-end latency, it decides to offload them fully onto the edge server, even when the server computation load is heavy. In the two cases, the performance of LoADPart keeps the same with the baseline, and hence we do not plot the baseline result in the corresponding subfigures in Figure 9. For ResNet18, we have already explained that choosing to conduct inference locally is its optimal solution in Section V-B. Thus, the variation of the computation load of the edge server does not affect the end-to-end latency, and we do not plot the baseline result, either. For ResNet50, LoADPart determines to conduct inference locally when the GPU utilization is above 100%(h) and to offload the DNNs fully onto the edge server when the GPU utilization is below 100%(l). Its end-to-end latency is generally close to that of the baseline in this situation.

In brief, the results in this subsection demonstrate the efficacy of LoADPart to adjust the partition point according to the computation load of the edge server. For some specific DNNs, LoADPart can efficiently and flexibly reduce the end-to-end latency when the edge server is busy.

VI. Related Work

In this section, we discuss two categories of prior work that are closely related to LoADPart, including the studies on DNN offloading and those on DNN execution time prediction.

Deep Neural Network Offloading: Neurosurgeon[4] is the fundamental work of DNN partial offloading. In this work, DNNs are partitioned and offloaded according to network latency, energy consumption and load of data centers in order to reduce the end-to-end inference latency. DeepWear[5] extends DNN offloading in the scenario of wearable devices and introduces a way to reduce the number of possible partition points to further reduce the overhead of DNN partition. DADS[14] proposes a partition decision algorithm that can deal with DNNs with DAG architectures. AAIoT [15] introduces a dynamic programming algorithm to divide DNNs into multiple parts and deploys them to multi-layered IoT architectures in order to minimize the overall system response time. IONN[16] presents an incremental vertical offloading approach when the parameters of a DNN are not uploaded to the server in advance. DDNN[17], SPINN[18], and Edgent[19] leverage the early-exit mechanism proposed in BranchyNet [20] in DNN offloading, but since the architectures of the original DNNs are modified, they require re-training. However, sometimes the

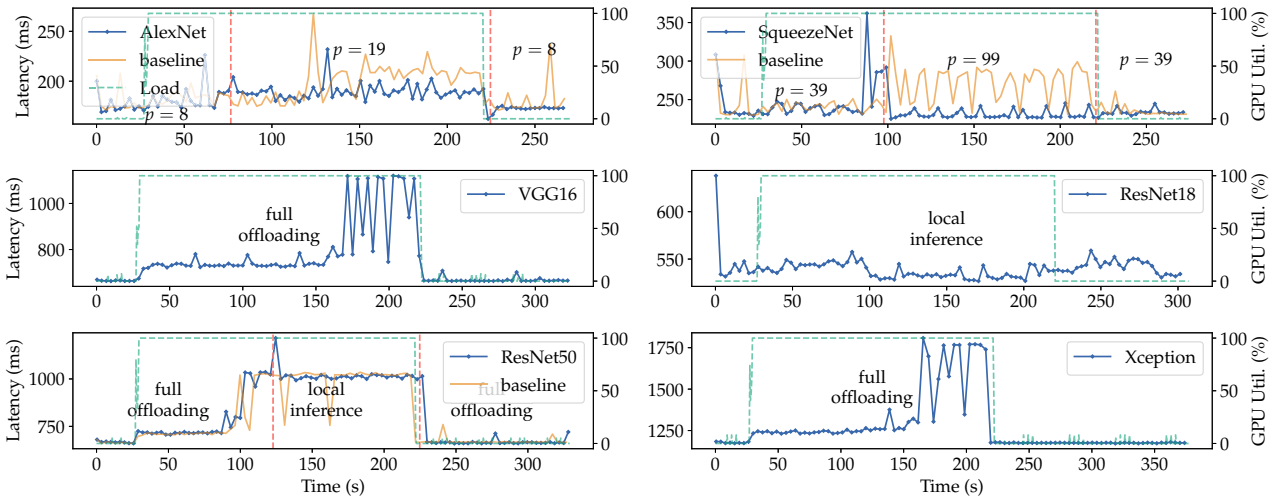

Fig. 9: The end-to-end latencies of the inference of AlexNet, SqueezeNet, VGG16, ResNet18, ResNet50, and Xception using LoADPart and the baseline (Neurosurgeon) with the variation of the server computation load.

user-end device may not have enough time and computation resources to perform re-training. In addition, the training data might be inaccessible if the DNN model is trained with proprietary datasets, which is quite common. It is worth noting that the previous work does not consider the computation load of the server seriously. Some of them mention the computation load of the cloud server and the mobile device respectively, but do not present detailed studies of how the computation load impacts the prediction of DNN execution time. Additionally, those algorithms that are able to cope with DNNs with DAG architectures incur high time complexity, which is not suitable for dynamic partition decision on resource-restricted devices. In this work, we propose a light-weighted partition decision algorithm that can process DNNs with DAG architectures, and clearly present how the computation load impacts the prediction of inference time as well as the partition decision.

Orthogonal to our work, MoDNN [21], DeepThings [22], and CoEdge [23] focus on partitioning computation inside each layer of a DNN and offloading the partitions to local worker nodes. They leverage parallelism inside each layer to speed up DNN inference.

Execution Time Prediction for Deep Neural Networks: Paleo[24] is the first work that models the execution time of DNNs. The execution time of a single neural network layer is estimated by the FLOPs and hardware speed provided by the manufacturer. In addition, the data transmission time is determined by the correspondent I/O bandwidth and data size. Justus et al.[25] suggest that the execution time of a single neural network layer is not strictly linearly related with its FLOPs. Thus, it leverages the power of a DNN to predict the execution time. The input features of this prediction DNN consist of layer specific features and hardware features. Habitat [26] makes performance predictions using the data collected from a GPU that the user already has. It makes predictions by scaling the execution time of each neural network layer from one GPU to another using either (1) wave scaling based on the GPU's execution model and (2) pre-trained multi-layer perceptrons (MLP) according to the GPU kernel implementations of layers. However, the prediction DNN makes the partitioning expensive in terms of resource consumption. To support light-weighted partition decision on the user-end devices, the time complexities of prediction models should be relatively low. NN-Meter [27] leverages random forests whose computation complexities are low to make predictions. In addition, it points out that due to the optimization conducted by inference frameworks, there exist fused neural network layers. It can detect the fused layers and make more precise predictions during runtime, instead of simply summing up single-layer predictions layer-by-layer, which may incur high prediction errors. In LoADPart, we study the computation characteristics of various typical kinds of DNN layers and carefully design the input features of LR models. The developed LR prediction models can achieve satisfactory accuracies with negligible overheads during the runtime. Though we do not encounter fused layers in our experiments, designing input features and training LR models for fused layers can be conducted by the same procedure with help of fused layer detection algorithms.

VII. Conclusion

In this paper, we propose LoADPart, a DNN partition and offloading system for edge intelligence. It dynamically determines the optimal partition point based on not only the network condition but also the server computation load, with a light-weighted partition decision algorithm integrating offline trained models and online monitored computation load to predict the end-to-end inference time with different partition decisions. We implement LoADPart for MindSpore, an emerging open-source DL framework. The evaluation on 6 DNNs

shows that LoADPart can minimize end-to-end inference latency with the improvement of 14.2% on average and up to 32.3% in some specific cases compared to the state-of-the-art solution, though for some DNNs the amount of improvement is limited due to the weak computing power of the tested user-end device. The improvement is typically equivalent to tens or hundreds of milliseconds, which can be significant to latency-sensitive interactive applications on edge devices.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their feedback and suggestions. This work is supported by NSFC funding No. 61922054, 62172270, 61832006 and 61803083, and SJTU-Huawei Innovation Research Lab Funding. Wenli Zheng and Li Li are the corresponding authors.

REFERENCES

[1] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[2] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html

[3] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *CoRR*, vol. abs/1710.09282, 2017. [Online]. Available: http://arxiv.org/abs/1710.09282

[4] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 615–629.

[5] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "Deepwear: Adaptive local offloading for on-wearable deep learning," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 314–330, 2020.

[6] mindspore ai, "Mindspore," 05 2021. [Online]. Available: https://github.com/mindspore-ai/mindspore

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[10] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794.

[11] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1800–1807.

[12] C. L. Lawson and R. J. Hanson, *Solving least squares problems*, ser. Classics in Applied Mathematics. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 1995, vol. 15.

[13] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[14] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive dnn surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1423–1431.

[15] J. Zhou, Y. Wang, K. Ota, and M. Dong, "Aaiot: Accelerating artificial intelligence in iot systems," *IEEE Wireless Communications Letters*, vol. 8, no. 3, pp. 825–828, 2019.

[16] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "Ionn: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 401–411.

[17] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.

[18] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: Synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20, 2020.

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

[20] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 2464–2469.

[21] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1396–1401.

[22] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[23] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2021.

[24] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=SyVVJ85lg

[25] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*. IEEE, 2018, pp. 3873–3882. [Online]. Available: https://doi.org/10.1109/BigData.2018.8622396

[26] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based computational performance predictor for deep neural network training," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 503–521. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/yu

[27] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, "Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–93. [Online]. Available: https://doi.org/10.1145/3458864.3467882