Joint Optimization of DNN Partition and Scheduling for Mobile Cloud Computing

Yubin Duan
Temple University
Philadelphia, USA
yubin.duan@temple.edu

Jie Wu
Temple University
Philadelphia, USA
jiewu@temple.edu

ABSTRACT

Reducing the inference time of Deep Neural Networks (DNNs) is critical when running time sensitive applications on mobile devices. Existing research has shown that partitioning a DNN and offloading a part of its computation to cloud servers can reduce the inference time. The single DNN partition problem has been extensively investigated recently. However, in real-world applications, a mobile device usually generates multiple DNN inference jobs simultaneously, and little attention has been paid to this case. We aim to minimize the makespan of multiple DNNs by jointly optimizing their partitioning and scheduling. Our observations show that the local computation time on a mobile device follows an increasing function, while the communication workload for offloading is usually decreasing as more DNN layers are computed. Based on this, we first relax our problem on continuous domain and show that partitioning all line-structure DNNs at the same layer is sufficient for makespan optimization. Then, for the discrete domain, two types of partitions are sufficient when the time difference between two adjacent partition layers is not drastic, subject to a given condition. An algorithm based on the binary search that efficiently finds optimal partition layers is illustrated. We also extend our approach to general-structure DNNs and offer a heuristic solution. Experiments have been conducted to evaluate the performance of different partition and scheduling methods on sample DNNs. Results validate the optimality of our theoretical results.

KEYWORDS

DNN partition, makespan minimization, mobile cloud offloading, pipeline, scheduling

ACM Reference Format:
Yubin Duan and Jie Wu. 2021. Joint Optimization of DNN Partition and Scheduling for Mobile Cloud Computing. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472468

1 INTRODUCTION

Deep Neural Networks (DNNs) have become increasingly popular in computer vision applications, such as image segmentation and recognition. These applications usually require efficient processing of DNN inference jobs, especially when the applications are running on mobile devices that have relatively weaker computational abilities compared to cloud servers. For example, when running augmented reality applications on smart glasses, the user experience heavily depends on the response time. Thus, it is critical to reduce the makespan of those DNN inference jobs.

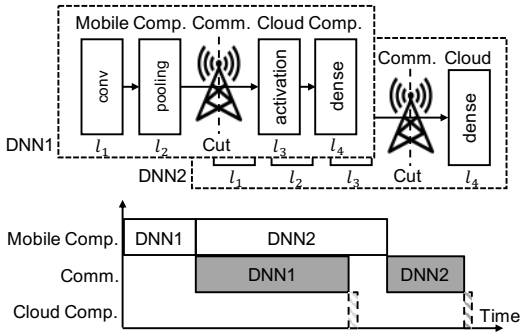


Figure 1: Overview of the DNN partition and scheduling.

Prior research has shown that an intelligent collaboration between a mobile device and the cloud server could reduce the latency of DNN inference jobs [10]. Partitioning the DNN computation between the mobile device and the cloud can reduce the makespan compared to mobile-only and cloud-only approaches [12]. The mobile-only approach refers to executing the DNN inference on a mobile device. The cloud-only approach refers to sending all data to the cloud and running the inference there. However, little attention has been paid to optimizing the multiple-job case.

In real-world applications, a mobile device usually generates multiple DNN inference jobs at the same time. Self-driving cars [16] and virtual/augmented reality [1, 14] are some typical examples. In those applications, the mobile end device needs to repeatedly run inference jobs on the same DNN. Specifically, self-driving cars usually equip multiple cameras and use the same DNN to process the frames collected from different cameras. Similar situations could also happen in virtual/augmented reality applications. Multiple image frames are generated at the same time. The processing procedures of those frames are usually the same meaning, they use the same DNN structure. Minimizing makespan of these jobs on the mobile device and the cloud by jointly partitioning and scheduling is very challenging, but its result is far reaching.

In this paper, we focus on optimizing the collaboration between the mobile device and the cloud when there are multiple homogeneous DNN inference jobs. Our objective is to reduce their makespan,

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472468

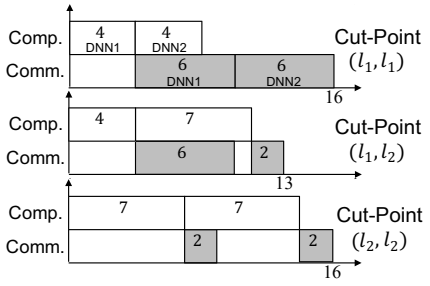


Figure 2: Partition and scheduling are correlated.

which is the length of the duration between the first job's starting time and the last job's completion. Fig. 1 illustrates the scheduling of a line-structure DNN with four layers (l_1, l_2, l_3, l_4). There are two major challenges to our optimization problem. The first one is how to partition multiple DNNs; is it optimal to partition all DNNs at the same layer? The other one is how to optimally schedule the execution of partitioned DNNs. Executing a partitioned DNN involves three steps: computing on the mobile device, offloading the intermediate result to the cloud, and computing the rest of the DNN on the cloud. Scheduling multiple DNNs with those execution steps can be categorized as a flow shop problem[2].

The above challenges can be revealed in a simple go-through example in Fig. 2. Assume we have two DNNs. Assume each DNN has three layers (l_1, l_2, l_3) with two partition (or cut) points: after l_1 and after l_2. Assume the local computation time on the mobile device for cutting after l_1 and l_2 are 4 and 7, respectively. The offloading time of partitioning after l_1 and l_2 are 6 and 2, respectively. The computation time on the cloud is negligible. This is because the computation power of the cloud server is usually much higher than mobile devices. If we partition both of the two DNNs after the first or the second layer (first or third case of Fig. 2), the makespan of the optimal schedule is 16. In contrast, if the two DNNs are partitioned at the first and second layers, respectively, then the optimal makespan is 13 (second case in Fig. 2). In this example, partitioning DNNs at different positions is a better choice. However, if we change the offloading time 7 to 5, the optimal partition changes. Partitioning both DNNs at the first layer becomes the optimal solution. How many types of partitions should we perform on n identical DNNs with k layers? Each DNN has $O(k)$ partition points, and checking all $O(k^n)$ combinations for n DNNs is computationally expensive.

We observe some useful properties of DNNs and analyze the optimal conditions of the partition problem. Our observation on commonly used line-structure DNNs shows that the offloading volume for mobile devices is usually decreasing as the partition layer moves down the linear layers. Although the volume of some intermediate layers may increase as the partition layer goes deep, we can cluster those layers as a virtual block without missing the optimal partition point. Offloading after some layers within the virtual block cannot be optimal because it is not wise to increase the offloading volume while running more computation workload on the mobile device. In addition, the offloading volume after each layer or virtual block usually decreases exponentially because DNNs need to quickly extract key features before classification. The computation time usually increases linearly since DNNs typically consist of many repeated blocks.

Based on these observations, we first relax our problem on continuous solution space and show that partitioning all DNNs at the same layer is sufficient for makespan optimization. Then, we expand our analysis to the discretized problem space. Although sharing the same partition layers on all DNNs is no longer optimal, we show that two types of partitions are sufficient when the time difference between two adjacent layers is not drastic. In addition, we illustrate an $O(\log k)$ algorithm based on the binary search to efficiently find optimal partition layers for line-structure DNNs with k layers. Additionally, we also extend our approach to general-structure DNNs and offer a heuristic solution.

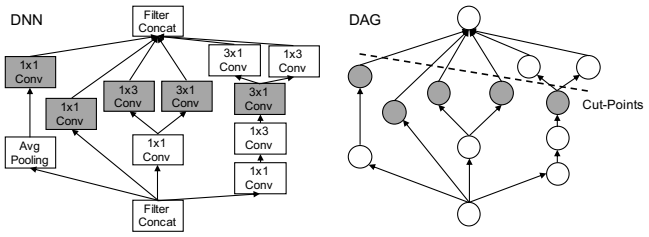
In the experiment, we test our joint optimization scheme with AlexNet, MobileNet, ResNet, and GoogLeNet, which are widely used in computer vision applications. The experiment results show that jointly considering the DNN partition and scheduling could further reduce the makespan of all jobs, compared to merely considering only DNN partition or only DNN scheduling.

Our contributions are summarized as followings:

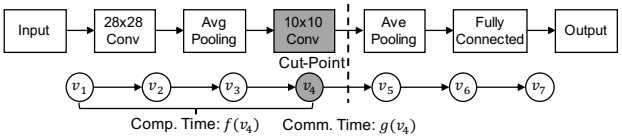
- We formulate a makespan minimization problem for multiple DNN inference jobs in mobile cloud computing. We jointly consider the DNN partition and scheduling.
- We theoretically analyze the partition and scheduling of line-structure DNNs. We reveal the optimal conditions for using one or two ways of partitions.
- We propose a binary-search-based algorithm to find the optimal cut-points for line-structure DNNs and extend it to deal with general-structure DNNs.
- Experiments on AlexNet, MobileNet, ResNet, and GoogLeNet, which are widely used in computer vision applications, show that our joint optimization scheme outperforms the scheme that only considers the DNN partition or scheduling.

2 RELATED WORK

Cloud/edge offloading approaches investigate the collaboration between local and remote computation resources. Neurosurgeon [10] proposed the idea of computation offloading for DNNs. The collaboration between mobile devices and the cloud server could speedup the DNN inference. However, Neurosurgeon only considers the partition of a single DNN. Our paper jointly considers the partition and scheduling of multiple DNNs. Teerapittayanon *et. al.* [24] proposed DDNN to reduce the communication data size in a distributed computing system containing mobile devices, the edge, and the cloud. Different from their objective, we aim to reduce the makespan that contains both communication and computation latencies. Mohammed *et. al.* [15] proposed to partition DNN into more than two parts to fit the fog computing scenario. Wang *et. al.* [28] presented an adaptive DNN partition scheme for inference acceleration. [3] further proposed an optimal partition algorithm for tree-structure DNNs. Although they considered multiple DNNs, the scheduling of multiple jobs is not discussed. In our paper, we allow DNN offloading stages to work in a pipeline. Carefully scheduling those jobs could further reduce their completion time.

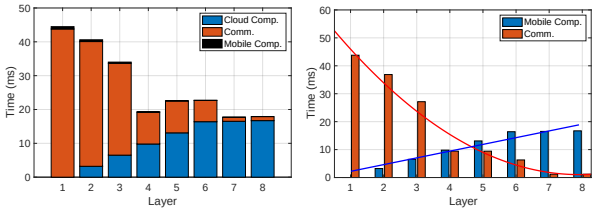


(a) General-structure DNN and its DAG representation



(b) Line-structure DNN and its DAG representation

Figure 3: DNNs and its DAG representations.



(a) Cloud comp. time is negligible (b) Trend of time consumption

Figure 4: Time consumption of each layer of AlexNet.

Other inference acceleration approaches include DNN model simplification [4, 6, 8, 23, 26, 27] and hardware acceleration [11, 25, 29, 30]. Our methods are compatible with those approaches. In addition, DNN model simplification accelerates an inference job by modifying and simplifying DNN models. This approach investigates the trade-off between computation workload and model accuracy. Our proposed methods do not modify the DNN model and do not sacrifice the inference accuracy, which is important in specific applications, such as autopilots.

3 MODEL

3.1 Problem Formulation

We use a Directed Acyclic Graph (DAG) to model a DNN. Formally, let $G = (V, E)$ denote a DAG, where V is the node set and E is the edge set. Each node $v \in V$ represents a layer in the DNN instead of a neuron since our partition granularity is layer-wise. A weighted edge $e \in E$ represents the data communication between two vertices that are incident to e. The edge weight shows the communication volume. Fig. 3(a) illustrates the structure of the inception-v4 network [21] and its DAG representation. Although a DNN could have a complex DAG representation, many widely-used DNNs are simple and have line structures as shown in Fig. 3(b). For example, VGG16[20], Tiny YOLOv2[17], and NiN[13] are commonly used in computer vision applications, and all have line-structure representations with different sizes.

Let J denote the set of DNN jobs, where j is used to index a job and n is the number of jobs. Each job j is a DNN inference task whose computation graph is G. Merely processing the job on mobile devices could be time consuming because of the weak computational power. A better approach is to offload a part of G to a cloud server[10]. The collaboration between mobile devices and the cloud contains three steps: 1) computing parts of G on mobile devices, 2) offloading intermediate results to the cloud server, and 3) computing the remaining parts of G on the cloud. The cloud server needs to send inference results back to mobile devices, but the communication volume is small and negligible. In this approach, the computation graph G of each job j is partitioned into two parts, which introduces the model partition problem. After partition, the mobile device needs to determine the processing sequence of partitioned graphs, which is the scheduling problem.

Although jobs have the same DAG structure, they could have different ways of partitioning. Let $P_j \subset V$ denote the partition of job j and be a set of cut-points. For line-structure DNNs, the partition set P_j only contains one cut-point. We can choose a partition layer and cut the DNN after the layer. For general-structure DNNs, the set P_j can include multiple cut-points. Specifically, all computation nodes $v \in P_j$ and their predecessors are processed on mobile devices. We use $f(P_j)$ to denote the time consumption of processing those nodes on the mobile device. The output of cut-points $v \in P_j$ needs to be offloaded to the cloud. The time consumption of sending the output is denoted as $g(P_j)$. The value of $f(P_j)$ and $g(P_j)$ could be predicted by using regression models[10]. The successors of cut-points $v \in P_j$ are computed on the cloud. The computation power of cloud servers is usually much larger than that of mobile devices. Therefore, the processing time of the cloud is negligible. Fig. 4(a) shows the time consumption of each step when partitioning AlexNet on different layers. The figure shows the cloud processing time is negligible. The notations are illustrated in Fig. 3(b), where v_4 is the cut-point. Nodes v_1, v_2, v_3, v_4 are processed on the mobile device and their time consumption is denoted as $f(v_4)$. The intermediate results generated by v_4 are sent to the cloud, and the communication time is denoted as $g(v_4)$.

After inference jobs are partitioned, the mobile device needs to schedule their processing sequence. For job j, the mobile device need to process the cut-points $v \in P_j$ and their predecessors, which is referred to the *computation stage* of j. After the computation stage is done, intermediate results are sent to cloud servers, which is denoted as the *communication stage* of j. When the computation (resp. communication) stage of job j starts, it acquires all computation (resp. network) resources. Otherwise, the makespan may be enlarged [19]. However, the computation and network resources could be used in a *pipelined* manner. The computation stage of a job can overlap with the communication stage of another job. The lengths of computation and communication stages of job j are $f(P_j)$ and $g(P_j)$, respectively. The communication stage cannot start until the corresponding computation stage is done. The completion time of job j is denoted as τ_j. All jobs in J are available at the time 0.

Our objective is to minimize the completion time or the makespan of n identical DNN inference jobs in J. The makespan is defined as

(a) Communication-heavy job set (b) Computation-heavy job set

Figure 5: Illustrations of sorted order sets in scheduling.

$\max_j \tau_j$. We jointly consider partitioning and scheduling strategies in our optimization problem.

3.2 Problem Analysis

The complexity of the problem mainly comes from the correlation between partitioning and scheduling. Specifically, the lengths of communication and computation stages of job j are functions of partition methods P_j. This indicates the potential collaboration between partitioning and scheduling. It is difficult to determine what is a good partition of a job, especially when the computation and network resources are used in a pipelined manner. The communication stage of a job could be completely hidden behind the computation stage of the next job, which helps to reduce the overall makespan. Therefore, it is necessary to jointly consider the partition of n jobs. However, examining all possible partitions of n DAGs cannot be done in polynomial time. Formally, assume each DAG has c ways of partitioning. The number of all possible partitions for n DAGs is $O(c^n)$. This leads to a natural question: do we need to investigate all possible partitions for n DAGs? What is the best partition strategy of each DAG when considering the potential pipelined speedup among multiple jobs?

After investigating some typical DNNs, we notice that f and g functions have useful monotonicity and convexity properties when the DAG has a line structure, which could help us answer those questions. Many widely used DNNs in computer vision applications have line structures, such as VGGNet[20] and YOLO[17]. The line structure makes partitioning easier since the partition P_j only contains one vertex, and both f and g become unary functions.

More importantly, if we index vertices in the DAG by their depths as shown in Fig. 3(b), then f is monotonically increasing and g is non-increasing. The computation time f increases as the partition layer moves forward because more layers need to be processed on the local mobile device. Admittedly, the communication time may increase as the partition layer goes deep. However, we can cluster the layers, after which the offloading volume increases, as a virtual block without missing the optimal partition point. Partition after any layer in the virtual block would enlarge the offloading communication volume and the local computation time. Therefore, it must not be the optimal partition point. We use the DAG in Fig. 3(b) to illustrate the monotone property. If the cut-point changes to v_5 instead of v_4, then the mobile device needs to process an additional average pooling layer that corresponds to v_5. At the same time, the additional average pooling layer could reduce the volume of the intermediate results, and the communication time could be reduced. Other types of layers such as convolutional layers or normalization layers would maintain the intermediate results size. Therefore, the communication workload is non-increasing. The monotonic properties of f and g are also revealed by Fig. 4(b). Notice that the layer shown in the figure represents a block of convolution, pooling, and activation operations. Based on the property, we could simplify the formulation of the makespan. The details are shown in Section 4.

Furthermore, we notice that the evolution of f and g function values w.r.t. the depth of cut-points could be fit by linear and convex functions, respectively. Typical line-structure DNNs are repeatedly built with units of convolutional and pooling layers. The duration of computing a unit is usually fixed. Therefore, the evaluation computation workload is nearly linearly increasing. Each unit can usually reduce the volume of intermediate results by half. Therefore, the function g fits in a convex function. By examining the convex properties, we find that partitioning all jobs in the same cut-point is sufficient for optimal scheduling when functions f and g are relaxed to continuous spaces $\mathbb{R}^+$. More details can be found in Section 5.

Algorithm 1 DNN Scheduling Algorithm

Input: Set of partitions $P = \{P_1, P_2, \ldots, P_n\}$.
Output: The optimal scheduling for the mobile device.
1: Evaluate $f(P_j), g(P_j)$ with regression models for each $j \in J$.
2: Communication-heavy set $S_1 \leftarrow \{j \in J | f(P_j) < g(P_j)\}$. Computation-heavy set $S_2 \leftarrow \{j \in J | f(P_j) \geq g(P_j)\}$.
3: $S_1 \leftarrow$ Sort S_1 for ascending order of $f(P_j)$.
4: $S_2 \leftarrow$ Sort S_2 for descending order of $g(P_j)$.
5: $S \leftarrow S_1 || S_2$.
6: **return** S as the optimal schedule.

4 DNN SCHEDULING

In this section, we first show an optimal scheduling algorithm for arbitrary partition strategies. Then, we focus on line-structure DAGs and formulate the optimal makespan, which could guide the partition strategy design.

4.1 Scheduling for Arbitrary Partitions

Scheduling for partitioned DAGs can be viewed as flow shop problems [2] and can be optimally solved by Johnson's rule [9]. Given the partition P_j for each job $j \in J$, we obtain the value of $f(P_j)$ and $g(P_j)$ by applying regression techniques [10]. With those known values, the scheduling problem can be categorized as a 2-stage flow shop problem. With an objective of minimizing the makespan of all jobs, Johnson's rule [9] can be applied to optimally solve the scheduling problem.

The procedures of the scheduling algorithm based on the Johnson's rule are illustrated in Alg. 1. Specifically, at line 1, we evaluate the values of $f(P_j)$ and $g(P_j)$ with regression models. Lines 2-5 show the procedures of Johnson's rule. Jobs in J are first split into two groups. The communication-heavy set S_1 contains all jobs whose communication stage is longer than the computation stage. The computation-heavy set S_2 contains the other jobs. Then, the jobs in the communication-heavy set are sorted based on the ascending order of their computation stage lengths. The jobs in S_1 are stored in increasing order of $f(P_j)$. Jobs in S_2 are sorted in descending order of $g(P_j)$. Illustrations of the sorted S_1 and S_2 are shown in Fig. 5. Finally, the sorted jobs in S_2 are concatenated after jobs in S_1, and S stores the optimal solution.

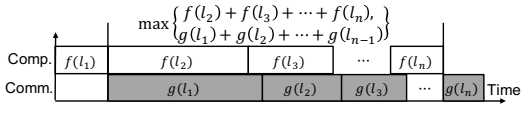


Figure 6: An illustration of makespan calculation.

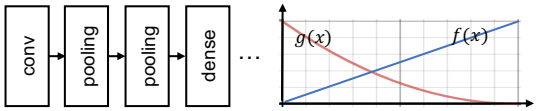


Figure 7: Continuous time consumption functions.

4.2 Makespan of Line-Structure DAG

For line-structure DAGs, the partition P only contains one element, after which the line-structure DAG is partitioned. Therefore, the functions f and g become unary functions in discrete domains. Let x_j denote the index of the cut-point for DAG j. Then, the makespan $\max_j \tau_j$ has a closed-form formulation. Concatenating the S_2 shown in Fig. 5(b) after the S_1 shown in Fig. 5(a) would cause idle time slots for either computation or communication resource but not both. Therefore, we have the following proposition.

Proposition 4.1. *For line-structure DAGs, if mobile devices schedule partitioned DAGs based on Johnson's rule, then the makespan of n jobs is* $\max_j \tau_j = f(x_1) + \max\{\sum_{i=2}^{n} f(x_i), \sum_{i=1}^{n-1} g(x_i)\} + g(x_n)$.

An illustration of the proposition is shown in Fig. 6. When n is large or $n \to \infty$, the makespan becomes large or $\max_j \tau_j \to \infty$. Therefore, it is more meaningful to investigate the average makespan $(\max_j \tau_j)/n$. To simplify the following analyses, we first rewrite the formulation of the average makespan

$$\lim_{n\to\infty} \frac{\max_j \tau_j}{n} = \lim_{n\to\infty} \frac{\max\{g(x_n)+\sum_{i=1}^{n} f(x_i), f(x_1)+\sum_{i=1}^{n} g(x_i)\}}{n}$$
$$= \lim_{n\to\infty} \max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$$

Then, the objective of our optimization problem is equivalent to $\min \max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$. Notice that n is a finite number in real-world applications, it is treated as a finite number in the following analysis. Nevertheless, the formulation of the average makespan is still a good approximation. Let k denote the length of the line-structure DAG, i.e., $k = |V|$. Use $l \in \{1, \ldots, k\}$ to index vertices in V from left to right (source to termination node). Our optimization problem becomes:

$$\text{P1: } \min \max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$$
$$\text{s.t. } \prod_{l=1}^{k} (x_j - l) = 0, \forall j \in J$$

To simplify the problem, we relax the domain of $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ to real numbers, i.e. $\mathbf{x} \in \mathbb{R}^n$. The relaxed problem becomes:

$$\text{P2: } \min \max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$$
$$\text{s.t. } x_j > 0, \forall j \in J$$

5 DNN PARTITION

In this section, we first discuss the optimal condition for line-structure DNN partition. Then, we introduce our partition and scheduling algorithms in detail.

5.1 Partition for Line-Structure DNNs

The objective of the partition is to minimize the makespan. We first analyze the relaxed problem P2 in continuous domains, then extend results to discrete domains.

In the continuous domain, we use convexity and monotonicity properties of functions f and g to analyze the optimal conditions of the problem. As shown in §3.2, the communication time $g(x)$ usually decreases exponentially along with x, and the computation time $f(x)$ usually increases linearly. Based on this observation, we assume f is an increasing linear (also convex) function and g is a decreasing convex function in the following analysis. The functions in the continuous domain are shown in Fig. 7.

When both f and g are convex, the problem P2 becomes an convex optimization problem. More interestingly, the problem holds a strong duality as shown in Lemma 5.1. It is mainly because that summation and the maximum of convex functions are still convex.

Lemma 5.1. *Our optimization problem P2 holds a strong duality if both $f(x)$ and $g(x)$ are convex.*

Proof: The objective function of P2 is convex when $f(x)$ and $g(x)$ are convex since the summation and the maximum of convex functions are still convex. The constraints are also convex. Therefore, P2 is a convex optimization problem.

The strong duality holds since our convex optimization problem satisfies the Slater's condition. Formally, we need to find a point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ in the feasible solution domain such that $x_i > 0, \forall i = 1, 2, \ldots, n$. Note that x_i represents the partition for job i. Such point exists since the partition of each job is independent and can be placed at any layer in middle of the DAG. ■

Especially, DNNs are mainly constructed by repeatedly placing blocks of convolution and pooling layers. The computation time of each block is similar. It makes the function $f(x)$ almost increase linearly with x. After each block, the sizes of intermediate results decrease exponentially because of the pooling layers. Even if the pooling layer is not inserted between two blocks, the size would not increase. Hence, $g(x)$ can be fit by a convex function.

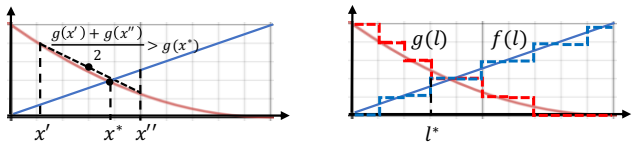
Because of the strong duality, we can find the optimal solution to the problem according to KKT conditions. As shown in Theorem 5.2, our analysis reveals an interesting property that all of n identical DNN inference jobs should be cut at the same point when we investigate the problem in the continuous domain.

Theorem 5.2. *After relaxing the partition point into a continuous space, partitioning all homogeneous line-structure DAGs at the same point could reach the optimal makespan.*

Proof. Before proceeding to further analysis, we smooth the max function by using the LogSumExp (LSE) function.

$$\max\{\sum_{i=1}^{n} f(x_i)/n, \sum_{i=1}^{n} g(x_i)/n\}$$
$$= \lim_{\alpha\to\infty} \frac{1}{\alpha} \ln\left(\exp(\alpha \sum_{i=1}^{n} f(x_i)/n), \exp(\alpha \sum_{i=1}^{n} g(x_i)/n)\right).$$

According to Lemma 5.1, the strong duality holds. Hence, the KKT conditions hold at the optimal point. Specifically, $\mathbf{x}^*$ is the optimal solution to the primal problem if and only if $\nabla_{\mathbf{x}} \lim_{\alpha\to\infty}$



(a) Continuous domain (b) Discrete domain

Figure 8: Graph explanation of the optimal partition.

$\frac{1}{\alpha}\ln\left(\exp(\frac{\alpha\sum_{i=1}^n f(x_i)}{n}), \exp(\frac{\alpha\sum_{i=1}^n g(x_i)}{n})\right) = 0$ and $x_i > 0, \forall i = 1, 2, \ldots, n$.

The gradient of the objective function is formed by a vector of partial orders of x_i. Formally, the partial order of each x_i is $f'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n f(x_i)\right) + g'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n g(x_i)\right)$.

For each x_i, at the optimal point, it satisfies

$$f'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n f(x_i)\right) = -g'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n g(x_i)\right) \quad (1)$$

In the continuous domain, the computational workload increases along with x, while the communication volume decreases. Therefore, $f'(x) > 0$ and $g'(x) < 0$. Then, $f'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n f(x_i)\right) > 0$ and $-g'(x_i)\exp\left(\frac{\alpha}{n}\sum_{i=1}^n g(x_i)\right) > 0$.

Take the logarithm of both sides of Eq. (1), we have:

$$\ln(f'(x)) + \frac{\alpha}{n}\sum_{i=1}^n f(x_i) = \ln(-g'(x)) + \frac{\alpha}{n}\sum_{i=1}^n g(x_i).$$

Rearrange terms in the previous equation, we have:

$$\sum_{i=1}^n (f(x_i) - g(x_i)) = (n/\alpha)\ln\left(-g'(x_i)/f'(x_i)\right) \quad (2)$$

When $\alpha \to +\infty$, the previous equation becomes $\sum_{i=1}^n (f(x_i) - g(x_i)) = 0$, since $-g(x_i)/f(x_i)$ is finite for $x_i > 0$. Let x^* denote the point such that $f(x^*) = g(x^*)$. If we set $x_i = x^*, \forall i = 1, 2, \ldots, n$, Eq. (2) holds for all $i = 1, 2, \ldots, n$. According to the KKT condition, $x_i = x^*, \forall i = 1, 2, \ldots, n$ is an optimal solution to our optimization problem. It shows that when both $f(x)$ and $g(x)$ are convex functions, partitioning multiple homogeneous DNNs at the same location can achieve the optimal result. ■

A graph explanation of the Theorem 5.2 is shown in Fig. 8(a). For any partition layer other than x^*, it enlarges either the communication or computation time. Besides, the increment of the communication or computation time cannot be averaged out by pairing with another job with a different cut-point. In the example, the average communication time of partitioning at x' and x'' is still larger than the optimal.

However, in real-world applications, the partition point is not continuous. The optimal solution we formulated in the previous part may not be able to be reached. In this case, partitioning all DAGs at the same point may no longer be optimal. Inspired by the optimal partition condition of continuous cases, we try to partition the discrete layers such that the difference between f and g values is small. In the discrete domain, f and g have discrete values as shown in Fig. 8(b). Let l denote the partition layer. $f(l)$ is still increasing along with l and $g(l)$ is non-increasing. Hence, the absolute difference between $f(l)$ and $g(l)$ first decreases along with l, then increases. To find the smallest absolute difference, we only need to find the left-most layer l^* such that $f(l^*) \geq g(l^*)$. If $f(l^*) = g(l^*)$, then cutting n identical DAGs after l^* gives the optimal makespan. To show it can optimize the makespan, the graph explanation for the continuous case can be directly applied. If $f(l^*) > g(l^*)$, cutting all DAGs at l^* is no longer optimal. We consider to use either $l^* - 1$ or l^* as the cut-point for a DAG. Theorem 5.3 shows that performing those two types of partitions is sufficient to minimize the makespan in certain scenarios.

Algorithm 2 Line-structure DNN Partition

Input: Line-structure DNNs with k layers.
Output: The partition layers of the DNNs and the ratio.

```
Estimate computation and communication time after partitioning of each layer f(l_i), g(l_i).
Initialize the partition points l = 1, r = k.
while l < r do
    mid ← ⌊(l + r)/2⌋.
    if f(mid) < g(mid) then
        l ← mid + 1.
    else
        r ← mid.
Ratio ← ⌊(f(l) − g(l))/(g(l − 1) − f(l − 1))⌋.
return l − 1, l, and Ratio.
```

Theorem 5.3. *When $f(l^* - 1) + f(l^*) = g(l^* - 1) + g(l^*)$ and $g(l^* - 1) = f(l^*)$, performing two types of partitions on different DNNs is sufficient to reach the optimal makespan.*

Proof. In the scenario, we partition half of DNNs after layer $l^* - 1$ and cut the other half after layer l^*. Note that l^* is the left-most layer such at $f(l^*) \geq g(l^*)$. DNNs partitioned after $l^* - 1$ belong to communication-heavy set S_1 in scheduling since $f(l^*-1) < g(l^*-1)$. Others belong to computation-heavy set S_2. After concatenate the sorted S_2 after S_1, the communication time is perfectly hidden after computation. Swapping a job in S_1 with another job which is partitioned after $l' < (l^* - 1)$ would enlarge the makespan. Although $f(l') < f(l^* - 1)$ after swapping, the communication time increases since $g(l') > g(l^* - 1)$ and it becomes the bottleneck. The increment on the communication time g cannot be hidden behind the computation. Hence, the makespan increases. Similarly, swapping a job in S_2 with another job which is partitioned after $l'' > l^*$ would enlarge the makespan. Besides, simultaneously performing the two swapping will not reduce the makespan since $g(l') + g(l'') \geq g(l^*) + g(l^* - 1)$ when $l' < l^* - 1 < l^* < l''$. If some of n DNNs are partitioned after layers other than $l^* - 1$ and l^*, its impact on scheduling can be reduced to one of the three cases mentioned above. None of them would reduce the makespan. Therefore, performing two types of partitions is sufficient. ■

To satisfy the condition mentioned Theorem 5.3, the difference between two adjacent partition layers cannot be drastic. Real-world applications usually do not satisfy the conditions. However, inspired by the Theorem 5.3, we attempt to reduce the makespan by reducing accumulated difference between computation and communication time. When performing two types of partitions, we can adjust the ratio between them to reduce the accumulated difference. Specifically, when $f(l^*-1)-g(l^*-1) \neq g(l^*)-f(l^*)$, the ratio between the number of DNNs partitioned after $l^* - 1$ with the number of DNNs partitioned after l^* should be $\lfloor(f(l^*)-g(l^*))/(g(l^*-1)-f(l^*-1))\rfloor$.

Algorithm 3 General-structure DNN Partition and Scheduling

Input: A general-structure DNN.
Output: The partition and scheduling of the DNN.
1: Convert the input into a DAG with independent paths.
2: Initialize the partition set $P \leftarrow \emptyset$.
3: **for** each path i in the DAG **do**
4: $v \leftarrow$ Find the cut-point for path i with Alg. 2.
5: $P = P \cup v$.
6: Schedule the independent paths with modified Alg. 1.

5.2 Binary-Search-Based Partition Algorithm

We propose a binary-search-based partition algorithm to efficiently find the cut-point for line-structure DNNs in the discrete domain.

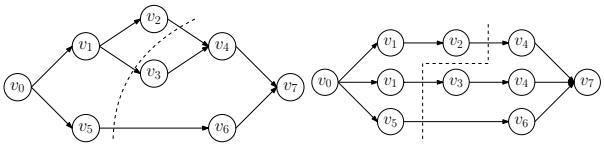
The steps of the partition algorithm are shown in Alg. 2. At line 1, we estimate the communication and computation time of each layer with linear regression models. The time consumption can be accurately estimated according to the layer type and shape as well as the network bandwidth[10]. After acquiring the values of functions f and g, we initialize two cut-points at line 2. k is the length of the line-structure DAG. Then, we iteratively update the cut-points by investigating the middle point mid of l and r. If $f(mid) < g(mid)$, the partition layer locates in the right side of mid. We update l into $mid + 1$ accordingly. Otherwise, the partition layer is left to mid and we update r into mid. The while loop terminates when $l = r$. The partition layer could either be l or $l - 1$. The ratio between the number of partition at $l-1$ and l is $\lfloor (f(l)-g(l))/(g(l-1)-f(l-1)) \rfloor$.

The correctness of the partition algorithm can be guaranteed by the loop invariant. In Alg.2, we can always guarantee that $f(l-1) < g(l-1)$ and $f(r) \geq g(r)$. Within the while loop, if the branch $f(mid) < g(mid)$ shown at line 5 is taken, then we have $f(l-1) < g(l-1)$ after l is updated by line 6. On the other hand, if the other branch shown at line 7 is taken, then $f(r) \geq g(r)$ is guaranteed after r is updated. When the while loop terminates, we have $l = r$. Taking those loop invariant properties into consideration, we have $f(l-1) < g(l-1)$ and $f(l) \geq g(l)$. The partition layer would be either l or $l-1$ since any further increments or decrements to l or $l-1$ would enlarge the difference between communication and computation time after partition. The complexity of the search algorithm is $O(\log k)$.

5.3 Partition for General-Structure DNN

In real-world applications, a DNN model may have more complex structures other than line-structures. The corresponding DAG contains multiple paths. A path is a sub-graph of the DAG which has a line-structure that starts from the input layer and ends at the output layer. For general-structure DAGs, the partition could spread across different paths as shown in Fig. 9(a). This provides more opportunities to fine-tune the length of the local computation and communication. However, the correlation among paths brings challenges for partition.

To decouple the correlation among paths, we convert the general DAG into a multi-path DAG structure without changing the partial order relations. An example of the conversion is shown in Fig. 9. Specifically, we convert each node in their topological orders. For a node, if its out-degree is larger than 1, then we duplicate the node based on its out-degree. Symmetric rules are applied to nodes whose in-degree is greater than 1.



(a) Before conversion (b) After conversion

Figure 9: An illustration of DAG conversion.

After the conversion, we focus on DAGs with multiple independent paths as shown in Fig. 9(b). Extensively exploring all possible combinations of cut-points in each path is computationally complex. We use a heuristic approach that partitions each path individually. For example, let there be 2 identical DAGs with structure as Fig. 9(a). They are converted into 2×3 individual paths, where 3 is the number of independent paths in each converted DAG. Procedures of the general-structure DNN partition and scheduling are shown in Alg. 3. Specifically, after converting the input DAG into a DAG with multiple individual paths at line 1, we initialize a cut-point set at line 2. Then, from line 3 to line 5, we find cut-points of different paths individually by using the searching algorithm illustrated in Alg. 2. After partition, we use Alg. 1 to schedule the execution of independent paths. Note that a slight modification of Alg. 1 is applied, i.e., duplicated nodes are only counted once when they are executed, although the Johnson's rule is applied to all nodes, including duplicated nodes, in determining the scheduling order.

Notice that our heuristic approach omits the potential collaboration opportunity between two different paths in the converted DAG. Overlapping sub-optimal partitions for the individual paths of a DAG may lead to an optimal schedule, which is worth further investigation in future work.

6 EXPERIMENT

6.1 System Setup

Our offloading system testbed consists of a mobile device and a cloud server. We use a Raspberry Pi model 4B as the mobile device and a PC in our lab as the cloud server. Specifically, the Raspberry Pi model 4B uses a quad-core Cortex-A72 (ARM v8) SoC as its CPU, and it has 4GB RAM. Our PC has a six-core Intel i7-8700 CPU with 32GB RAM and a GTX1080 GPU. The operating system installed on the PC is Ubuntu 20.04. The communication channel between the mobile and cloud device is set up within a wireless LAN based on Wi-Fi. To simulate the communication delay at different bandwidths, we use the `wondershaper` package to limit the upload and download bandwidth of the Raspberry Pi.

The prototype of our joint optimization system is implemented with Python. The client-side is running on the Raspberry Pi and the server-side is running on the PC. Both client and server use `PyTorch` as their machine learning engine to perform DNN inferences. DNN models used in our experiment are pre-cut at all possible partition points and initialized for the client and server. The server runs all inference tasks on its GPU with `CUDA`. The network communication between the client and server is established with `gRPC`. In a round

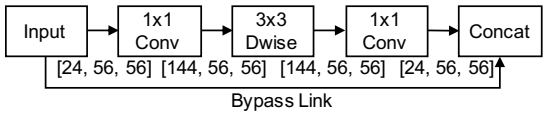


Figure 10: A bottleneck residual module in MobileNet. (The 3-tuple under each link shows the shape of the tensor transmitted between layers.)

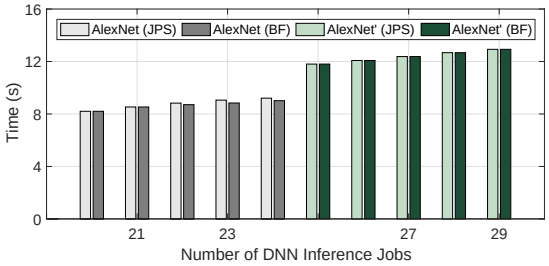


Figure 11: Compare with brute force search.

of DNN inference task, the client first loads the input image, transforms it to a tensor, and performs the forward propagation on the partitioned DNN. Then, the client collects the output tensor and encodes it as serialization for network transmission. The serialization is done by calling the `tensor.save()` method and the encoded tensor is flushed into a `BytesIO` which is a virtual interface in memory. Then, a gRPC client is called to collect bytes array through the `BytesIO` and sent it to server a gRPC request message. At the server-side, the received message is loaded from the `BytesIO` and decoded by calling the `tensor.load()` method. Then, a forward propagation using the decoded tensor on the remaining DNN is performed. Finally, the server sends the classification result back to the client via a gRPC reply message.

Our scheduler is implemented on the mobile device. Before partitioning and scheduling, the scheduler needs to estimate the computation time of local DNN inference and the communication time of offloading. To reduce the estimation overhead, we build a lookup table for computation time considering the local computation time stable. The size of the lookup table is limited since the number of commonly used DNN types is limited. The communication time changes with network bandwidth. Therefore, we use a simple linear regression model to estimate the communication delay. Specifically, the communication time $t_n = w_0 + w_1 \cdot r$, where w_0, w_1 are regression parameters and $r = s/b$ is the ratio of the message size s to the bandwidth d. w_0 represents the latency of setting up the communication channel. The lookup table is pre-built and the parameters of the regression model are pre-trained. The scheduler would load them into memory when starting.

We use `PyTorch Profiler` to measure the performance of DNN inference on the client and server when building the lookup table. To measure the communication delay, the gRPC reply message contains a field to record the total computation time t_c of the cloud server. The client will start a timer when it sends the gRPC request message, and stop the timer when it received the reply message. The duration of timer t_d includes the communication delay and the cloud computation delay. The difference $t_d - t_c$ is the communication delay. Our preliminary experiment result shows that the cloud computation delay is usually much smaller than the communication delay. Therefore, our scheduler only considers a two-stage scheduling problem and uses the time duration of t_d to train the regression model for communication delay.

In the experiments, we validate proposed algorithms on different types of DNNs which are widely used in CV applications. For the line architecture, we use AlexNet [5] and MobileNet-v2 [18]. It is important to mentioned that the MobileNet contains multiple *bottleneck residual modules* as shown in Fig. 10. The bottleneck residual module is a variant of the residual block that indents to create a bottleneck with 1×1 convolutions [5]. The 3-tuple under each link shows the shape of the tensor transmitted between layers. There is a bypass link in the module. Considering the bypass link, the MobileNet does not have a line-structure. However, from Fig. 10, we notice that the output sizes of layers within a bottleneck residual module are not decreasing. Partitioning at a layer within the module does not bring benefits for scheduling, and it should be clustered as a virtual block according to our analysis in §3.2. After clustering and converting, we treat the MobileNet as a line-structure DAG. For the general architecture, we use GoogLeNet [22]. GoogLeNet contains several Inception modules illustrated in Fig. 3(a). The Inception module should not be clustered as a virtual block, because the output tensor sizes of its intermediate layers are smaller than input tensor sizes. We treat the GoogLeNet as a general-structure DAG.

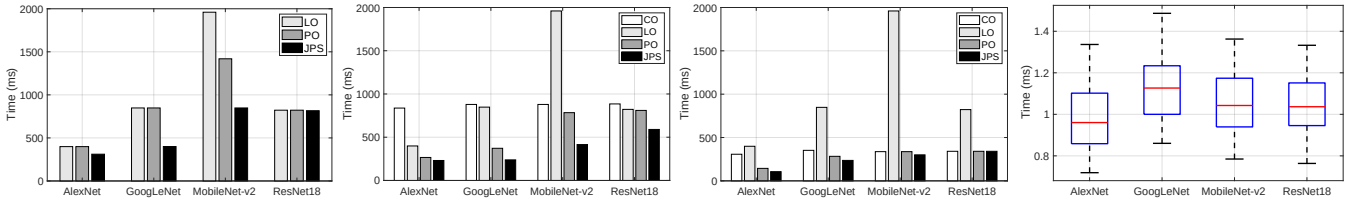
6.2 Comparison Algorithms

We compare our scheme that jointly considers the partition and schedule (which is denoted as JPS), with *partition only* (PO), *cloud only* (CO), and *local only* (LO) schemes. For PO, we implement the state-of-the-art DNN partition algorithm[7], which generates homogeneous cut-points for all jobs. However, this scheme does not consider the collaboration between partitioning and scheduling. For CO, the entire inference workload is done at the cloud server. The local mobile device upload all input tensors to the server. For LO, the inference jobs are processed merely on mobile devices without offloading. In addition, we implement the brute-force (BF) approach to find the optimal partition and schedule for small size inputs.

6.3 Experiment Results

We first compare our JPS with the BF approach to show the gap between our schedule with the optimal one. Fig. 11 shows the overall time consumption of multiple DNN inference jobs. In AlexNet, our scheme could generate optimal scheduling when the number of identical jobs is less than 23. On a synthetic DNN AlexNet′, whose communication time is sampled from the fitted curved, our scheme could find the optimal schedule. These experiment results verify that if the conditions stated in Theorem 2 holds, our scheme could find the optimal schedule for multiple identical DNNs.

We then evaluate our algorithms on line-structure and general-structure DAGs. In this experiment, we generate 100 repeated jobs for each type of DNN, and we record the average completion time over different bandwidths. Specifically, we choose three typical



(a) 3G (1.1 Mbps) (b) 4G (5.85 Mbps) (c) Wi-Fi (18.88 Mbps) (d) JPS overhead

Figure 12: Comparison of total inference latency.

Table 1: Latency reduction ratio compared with LO (%)

Model	3G		4G		Wi-Fi	
	PO	JPS	PO	JPS	PO	JPS
AlexNet	0	**22.06**	33.33	**42.11**	63.91	**73.43**
MobileNet-v2	27.60	**56.73**	60.00	**78.83**	82.81	**84.69**
GoogLeNet	0	**52.83**	56.13	**71.93**	66.63	**72.17**
ResNet18	0	**0.73**	1.46	**28.22**	58.52	**58.52**

bandwidths to simulate 3G, 4G, Wi-Fi network conditions. According to [7], the typical bandwidths of 3G, 4G, and Wi-Fi are 1.1Mbps, 5.85Mbps, and 18.88Mbps, respectively. The experiment results are shown in Fig. 12. In general, we can see that our joint optimization scheme JPS has the best performance for all types of DNNs in all network environments. Over each bandwidth configuration, our scheme that jointly considers the partition and schedule outperforms the other comparison algorithms. The PO scheme ignores the collaboration among multiple jobs in scheduling, while the LO scheme does not make use of the powerful cloud.

Fig. 12(a) illustrates the performance comparison on the 3G network. The CO time is not shown in the figure because it costs more than 4,000ms to upload the input tensor into the cloud server for all DNNs. It is much larger than other offloading schemes. From the figure, we notice that the JPS would significantly reduce the inference time for AlexNet, GoogLeNet, MobileNet. Especially for GoogLeNet, the JPS reduce the inference time by 52.5% compared with LO and PO. The improvement of JPS for ResNet is not obvious. It is because the network speed is too slow and offloading the intermediate result of any layer of ResNet would cost more time than compute the model locally.

When the network bandwidth increased to 5.85 Mbps, our JPS scheme achieves significant improvement for all DNNs used in our experiment. Compare Fig. 12(a) and Fig. 12(b), we can notice that the state-of-the-art PO algorithm can barely reduce the total inference time for ResNet, even the network condition is improved from 3G to 4G. Without scheduling, the bandwidth improvement is wasted. In contrast, our JPS would make full use of the bandwidth increase and reduce the overall inference time by 27.2% compared to PO. The reduction ratio of the inference time compared with LO is summarized in Table 1. Fig. 12(c) shows the performance comparison on the Wi-Fi network. The bandwidth of Wi-Fi is large and simply offloading all computation workload to the cloud server is a good strategy. In this situation, our JPS still could reduce the inference time for AlexNet, GoogLeNet, and MobileNet. Fig. 12(d) shows the overhead of our JPS scheme. From the figure, we notice that the overhead is negligible compared with the inference time. It is because both binary search and scheduling algorithms are fast. More importantly, we use a lookup table to store the local inference time. It saves the time cost of profile estimation. In addition, the communication time is estimated by using a simple linear regression model which is also time-efficient.

From Fig. 12, we notice that there is a range in which our JPS scheme can reduce the over inference time. When the network condition too poor, offloading brings no benefits. Similarly, when the bandwidth is large enough, the mobile device should simply upload all computation workload to the cloud server which is much faster. It is interesting to investigate the benefit range in which JPS can reduce the DNN inference time for different types of DNNs. The benefit range of AlexNet and MobileNet is shown in Fig. 13.

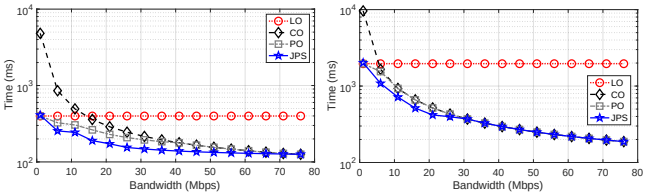
Fig. 13 shows the DNN inference time under different bandwidths. From the figure, we find that our JPS scheme can speedup both AlexNet and MobileNet in bandwidth range of $[1, 20]$ Mbps, which covers the bandwidth from 3G network to Wi-Fi network. Compare Fig. 13(a) and Fig. 13(b), we notice that the AlexNet has a wider benefit range in which JPS can reduce the inference time. It shows that even wireless upload bandwidth exceeds 50Mbps, our JPS scheme is useful.

We also investigate the impact of the ratio between computation- and communication-heavy jobs. The results are shown in Fig. 14. From Fig. 14(a), we can see that the optimal ratio between two types of jobs is not 1, and it varies with the bandwidth configurations. Comparing Fig. 14(a) and Fig. 14(b), we notice that if the communication-heavy jobs have larger differences between computation and communication stages, then the optimal ratio between computation-heavy and communication-heavy jobs is low. Otherwise, there should be more communication-heavy jobs. The optimal ratio shifts with bandwidth configurations.

Above all, experiment results show that jointly considering the DNN partitioning and scheduling could help to further reduce the makespan. If we only consider the DNN partition problem, the potential collaboration between different DNN inference jobs is ignored. The speedup brought by pipeline might be wasted. The improvement of the JSP varies with different bandwidth. It is more obvious when the bandwidth is limited.

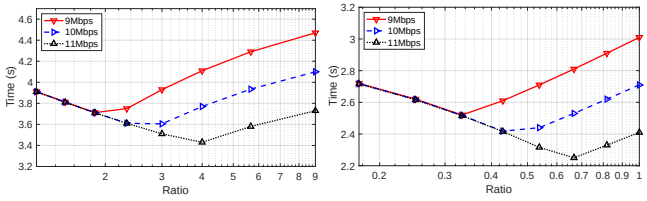
7 CONCLUSION

In computer vision applications, a mobile device usually generates multiple DNN inference jobs at the same time. Those DNNs usually



(a) AlexNet (b) MobileNet-v2

Figure 13: Inference latency under different bandwidths.



(a) ResNet (b) GoogLeNet

Figure 14: The impact of the ratio between two types of jobs.

have homogeneous structures. We consider that the computation power of mobile devices is usually weak. Offloading parts of the DNN to a cloud server could speed up the DNN execution. Mobile devices process the partitioned DNN with computation and communication stages. The partition strategies affect the length of each stage. Our objective is to minimize the makespan of all inference jobs. The makespan is correlated with partition and scheduling. We jointly consider the partition and scheduling problem. Particularly for line-structure DNNs, we find that if we relax the problem to continuous solution spaces, then partitioning all DNNs at the same point is sufficient for makespan optimization. Then, on the discretized problem space, two types of partitions are sufficient when the time difference between the two adjacent partition points is not drastic. We propose a binary-search-based algorithm to efficiently find the optimal partition points. Experiments on real-world CV applications show that our joint optimization scheme outperforms the scheme that only considers the DNN partition or scheduling. Joint partition and scheduling for jobs with multiple individual paths or heterogeneous jobs is worth further investigation.

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation grants CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, CNS 1651947, and CNS 1564128.

REFERENCES

[1] O. Akgul, H. Penekli, and Y. Genc. 2016. Applying Deep Learning in Augmented Reality Tracking. In *Proceedings of the IEEE SITIS*. 47–54.
[2] Peter Brucker and P Brucker. 2007. *Scheduling algorithms*. Vol. 3. Springer.
[3] Yubin Duan and Jie Wu. 2021. Computation Offloading Scheduling for Deep Neural Network Inference in Mobile Computing. In *IEEE/ACM 29th International Symposium on Quality of Service (IWQoS'21)*. Virtual Conference.
[4] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the ACM MobiSys*. 123–136.
[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE CVPR*. 770–778.
[6] 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. *arXiv preprint arXiv:1704.04861* (2017).
[7] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *IEEE INFOCOM*. 1423–1431.
[8] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. 2018. YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In *Proceedings of the IEEE Big Data*. IEEE, 2503–2510.
[9] Selmer Martin Johnson. 1954. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly* 1, 1 (1954), 61–68.
[10] 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. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.
[11] 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 ACM/IEEE IPSN*. 1–12.
[12] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. 2020. SPINN: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the ACM MobiCom*. 1–15.
[13] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).
[14] Chen Liu, Kihwan Kim, Jinwei Gu, Yasutaka Furukawa, and Jan Kautz. 2019. PlaneRCNN: 3D Plane Detection and Reconstruction From a Single Image. In *Proceedings of the IEEE CVPR*.
[15] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. 2020. Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading. In *Proceedings of the IEEE INFOCOM*. 854–863.
[16] Arsalan Mousavian, Dragomir Anguelov, John Flynn, and Jana Kosecka. 2017. 3d bounding box estimation using deep learning and geometry. In *Proceedings of the IEEE CVPR*. 7074–7082.
[17] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242* (2016).
[18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE CVPR*. 4510–4520.
[19] Wujie Shao, Fei Xu, Li Chen, Haoyue Zheng, and Fangming Liu. 2019. Stage Delay Scheduling: Speeding up DAG-style Data Analytics Jobs with Resource Interleaving. In *Proceedings of ICPP*. 1–11.
[20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[21] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of AAAI*.
[22] 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 CVPR*. 1–9.
[23] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive selection of deep learning models on embedded systems. *arXiv preprint arXiv:1805.04252* (2018).
[24] 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 IEEE ICDCS*. 328–339.
[25] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. (2011).
[26] Ehsan Variani, Xin Lei, Erik McDermott, Ignacio Lopez Moreno, and Javier Gonzalez-Dominguez. 2014. Deep neural networks for small footprint text-dependent speaker verification. In *Proceedings of the 45th IEEE International Conference on Acoustics, Speech, and Signal Processing*. 4052–4056.
[27] Can Wang, Sheng Zhang, Yu Chen, Zhuzhong Qian, Jie Wu, and Mingjun Xiao. 2020. Joint Configuration Adaptation and Bandwidth Allocation for Edge-based Real-time Video Analytics. In *Proceedings of the IEEE INFOCOM*. 1–10.
[28] Ning Wang, Yubin Duan, and Jie Wu. 2021. Accelerate Cooperative Deep Inference via Layer-wise Processing Schedule Optimization. In *IEEE ICCCN*. 1–9.
[29] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the ACM/EDAC/IEEE DAC*. 1–6.
[30] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An automated dnn chip predictor and builder for both FPGAs and ASICs. In *Proceedings of ACM/SIGDA FPGA*. 40–50.