# Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge

Chuang Hu*, Wei Bao†, Dan Wang*, Fengming Liu‡
*Department of Computing, The Hong Kong Polytechnic University
†School of Computer Science, The University of Sydney
‡Department of Electronic and Information Engineering, The Hong Kong Polytechnic University

***Abstract*—Recent advances in deep neural networks (DNNs) have substantially improved the accuracy and speed of a variety of intelligent applications. Nevertheless, one obstacle is that DNN inference imposes heavy computation burden to end devices, but offloading inference tasks to the cloud causes transmission of a large volume of data. Motivated by the fact that the data size of some intermediate DNN layers is significantly smaller than that of raw input data, we design the DNN surgery, which allows partitioned DNN processed at both the edge and cloud while limiting the data transmission. The challenge is twofold: (1) Network dynamics substantially influence the performance of DNN partition, and (2) State-of-the-art DNNs are characterized by a directed acyclic graph (DAG) rather than a chain so that partition is greatly complicated. In order to solve the issues, we design a Dynamic Adaptive DNN Surgery (DADS) scheme, which optimally partitions the DNN under different network condition. Under the lightly loaded condition, DNN Surgery Light (DSL) is developed, which minimizes the overall delay to process one frame. The minimization problem is equivalent to a min-cut problem so that a globally optimal solution is derived. In the heavily loaded condition, DNN Surgery Heavy (DSH) is developed, with the objective to maximize throughput. However, the problem is NP-hard so that DSH resorts an approximation method to achieve an approximation ratio of 3. Real-world prototype based on self-driving car video dataset is implemented, showing that compared with executing entire the DNN on the edge and cloud, DADS can improve latency up to 6.45 and 8.08 times respectively, and improve throughput up to 8.31 and 14.01 times respectively.**

## I. Introduction

Recent advances in deep neural networks (DNN) have substantially improve the accuracy and speed of computer vision and video analytics, which creates new avenues for a new generation of smart applications. The maturity of cloud computing, equipped with powerful hardware such as TPU and GPU, becomes a typical choice for such kind computation intensive DNN tasks. For example, in a self-driving car application, cameras continuously monitor and stream surrounding scene to servers, which then conduct video analytic and feed back control signals to pedals and steering wheels. In an augmented reality application, a smart glass continuously records its current view and streams the information to the cloud servers, while the cloud servers perform object recognition and send back contextual augmentation labels, to be seamlessly displayed overlaying the actual scenery.

One obstacle to realizing smart applications is the large amount of data volume of video streaming. For example, Google's self-driving car can generate up to 750 megabytes of sensor data per second [1], but the average uplink rate of 4G, fastest existing solution, is only 5.85Mbps [2]. The data rate is substantially decreased when the user is fast moving or the network is heavily loaded. In order to avoid the effect of network and put the computing at the proximity of data source, edge computing emerges. As a network-free approach, it provides anywhere and anytime available computing resources. For example, AWS DeepLens camera can run deep convolutional neural networks (CNNs) to analyze visual imagery [3]. Nevertheless, edge computer themselves are limited by their computing capacity and energy constraints, which cannot fully replace cloud computing.

From Fig. 1, we observe that, for the DNN, the amount of some intermediate results (the output of intermediate layers) are significantly smaller than that of raw input data. For example, the input data size of tiny YOLOv2 [4] is 0.95MB, while the output data size of intermediate layer max5 is 0.08MB with a reduction of 93%. This provides the opportunity for us to take the advantages of the powerful computation capacity of the cloud computing and the proximity of the edge computing. More specifically, we can compute a part of DNN on the edge side, transfer a small number of intermediate results to the cloud, and compute the left part on the cloud side. The partition of DNN constitutes a tradeoff between computation and transmission. As shown in Fig. 2, partition at different layers will cause different computation time and transmission time. So, an optimal partition is desirable.

Unfortunately, the decision on how to split the DNN layers heavily depends on the network conditions. In a LTE network, the throughput can decrease by 10.33 times during peak hours [5], and this value could reach 18.65 for a WiFi hotspot [6]. Under a high-throughput network condition, computing delay dominates and it is more desirable to offload the DNNs as early as possible. However, if the network condition degrades severely, we should prudently determine the DNN cut so to decrease the volume of data transmission. For example, Fig. 3 shows that when the network capacity is as high as 18Mbps, the optimal cut is at input layer and the overall processing delay is 0.59s. However, when the network capacity is lowered to 4Mbps, cutting at input layer is no longer valid as the communication delay increases substantially. Under this scenario, cutting at max5 is optimal, with a delay reduction of 62%.

Another challenge in the partition is that the recent advances of DNN show that DNNs are no longer limited to a chain topology, DAG topologies gain popularity. For example, GoogleNet [7] and ResNet [8], the champion of ImageNet Challenge 2014 and 2015 respectively, are DAGs. Obviously, partitioning DAG instead of chain involves much more complicated graph theoretic analysis, which may lead to NP-hardness

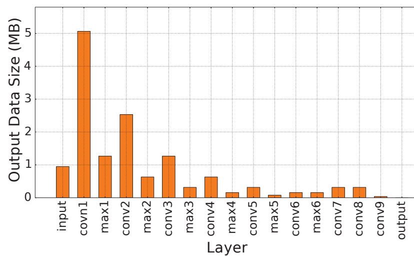


Fig. 1: The output data size of each layer of YOLOv2.

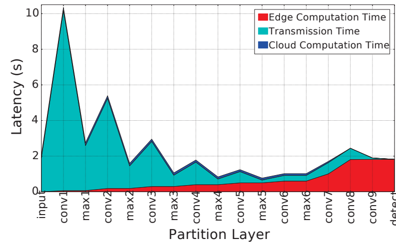


Fig. 2: Latency constitution when partition at the different layers of tiny YOLOv2. Bandwidth is $4$Mbps.

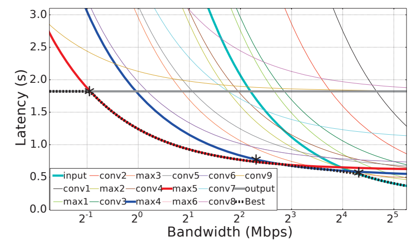


Fig. 3: The latency of partition at different layers of YOLOv2 as a function of bandwidth.

in performance optimization.

To this end, in this paper, we investigate the DNN partition problem, in order to find the optimal DNN partitioning in an integrated edge and cloud computing environment with dynamic network conditions. We design a Dynamic Adaptive DNN Surgery (DADS) scheme, which optimally partitions the DNN network by continually monitoring the network condition. The key design of DADS is as follows. DADS keeps monitoring the network condition and determines if the system is operated in the lightly loaded condition or heavily loaded condition. Under the lightly loaded condition, DNN Surgery Light (DSL) is developed, which minimizes the overall delay to process one frame. In this part, in order to solve the delay minimization problem, we convert the original problem to an equivalent min-cut problem so that the globally optimal solution can be found. In the heavily loaded condition, DNN Surgery Heavy (DSH) is developed, which maximizes the throughput, i.e. the number of frames can be handled per unit time. However, we prove such optimization problem is NP-hard, which cannot be solved within polynomial computational complexity. DSH resorts an approximation approach, which achieves an approximation ratio of 3.

Finally, we develop a real-world testbed to validate our proposed DADS scheme. The testbed is based on the self-driving car video dataset and real traces of wireless network. We test 5 DNN models. We observe that compared with executing entire DNNs on the cloud and on the edge, DADS can reduce execution latency up to 6.45 times and 8.08 times respectively, and improve throughput up to 8.31 times and 14.01 times respectively.

## II. An Edge-Cloud DNN Inference (ECDI) Model

### A. Background

Video analytics is the core to realize a wide range of exciting applications ranging from surveillance and self-driving cars, to personal digital assistants and automatic drone controls. The current state-of-the-art approach is to use a deep neural network (DNN) where the video frames are processed by a well-trained constitutional neural network (CNN) or recurrent neural network (RNN). Video analytics use DNNs to extract features from input frames of the video and classify the objects in the frames into one of the predefined classes.

DNN network consists of quite a few layers which can be organized in a directed acyclic graph (DAG). Fig. 4 shows a 7-layer DNN model. Inference for video is performed with a DNN using a feed-forward algorithm that operates on each frame separately. The algorithm begins at the input layer and progressively moves forward layer by layer. Each layer receives the output of prior layers as the input, performs a series of computation on the input data to get the output, and feeds its output to the successor layers. This process terminates once the computation of output layer is finished.

The video is generated at the edge side and the frames of the video are fed into the DNN as input. The computation of each layer in DNN can be performed at the edge or at the cloud. Computing layers at edge devices does not require to transmit data to the cloud but incurs more computation due to resource-constrained device. Computing layers at the cloud leads to less computation but incurs transmission latency for transmitting data from edge devices to the cloud.

### B. The ECDI Model

In this subsection. We formally present the ECDI model.

*1) Video Frame:* A video consists of a sequence of frames (pictures) to be processed, with a sampling rate $Q$ frames/second. Each sampled frame is fed to a predetermined DNN for inference. Please note that the sampling rate is not the frame rate of the video. It indicates how many frames/pictures are processed each unit time [9].

*2) DNN as a Graph:* A DNN is modeled as a directed acyclic graph (DAG). Each vertex represents one layer of the neural network. A layer is indivisible and must be processed on either the edge side or the cloud side. We add an virtual entry vertex and an exit vertex to represent the starting point and the ending point of DNN respectively. The links[1] represent communication and dependency among layers.

Let $\mathcal{G} = (\mathcal{V} \bigcup \{e, c\}, \mathcal{L})$ denote the DAG of DNN, where $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ is the set of vertices representing the layers of the DNN (specially, $v_1$ and $v_n$ represent the input layer and output layer respectively). $e$ and $c$ denote virtual entry and exit vertices (to facilitate the subsequent analysis). $\mathcal{L}$ is the set of links. A link $(v_i, v_j) \in \mathcal{L}$ represents that $v_i$ has to be processed before $v_j$, and $v_i$ feeds its output to $v_j$. Fig. 6 shows the DAG of the pure inception v4 network [10] in Fig. 5.

Since each layer can be processed on either the edge or cloud side, its processing time depends on where it is processed (i.e. on the edge or on the cloud). Let $t_i^e$ and $t_i^c$ be the time needed to process $v_i$ one edge and cloud respectively. Let $d_i$ and $t_i^t$ denote the output data size and the transmission time of $v_i$. We

[1] Please note that to avoid misunderstanding, throughout this paper, we use the term "link" to represent "edge of a graph." This is because "edge" in this paper has already represented "edge computing."

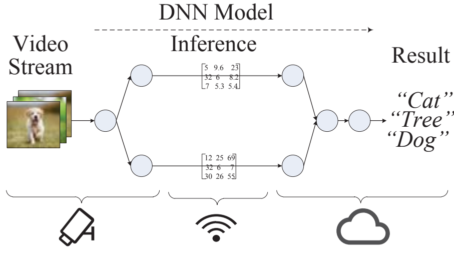

Fig. 4: A 7-layer DNN model classifies frames of video.

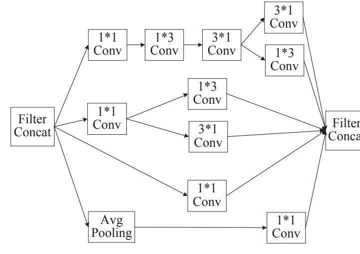

Fig. 5: The inception v4 network represented in layer form.

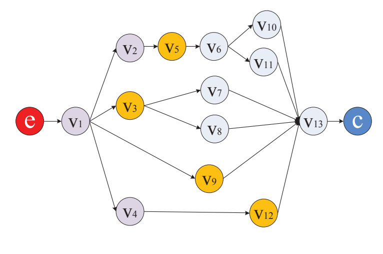
Fig. 6: Graph representation of inception v4 network.

define $\mathbf{D}_t = \{d_1, d_2, \cdots, d_n\}$. Let $B$ be the network bandwidth, we have $t_i^t = \frac{d_i}{B}$. Please note that $B$ can be dynamically changed and we need to adapt such changes. We define $\mathbf{F}_e = \{t_1^e, t_2^e, \cdots, t_n^e\}$, $\mathbf{F}_c = \{t_1^c, t_2^c, \cdots, t_n^c\}$, $\mathbf{F}_t = \{t_1^t, t_2^t, \cdots, t_n^t\}$. They denote the *three key delays*: processing delay at the edge, transmission delay, and processing delay at the cloud of each layer.

*3) DNN Partitioning:* Our objective is to partition DNN into two parts so the one part is processed at the edge and the other is processed at the cloud. Mathematically, we should find a set of vertices $\mathcal{V}_S$ as a subset of $\mathcal{V}$ such that removing $\mathcal{V}_S$ causes that the rest of $\mathcal{G}$ becomes two disconnected components. One component contains $e$, denoted by $\mathcal{V}_E'$ and the other component contains $c$, denoted by $\mathcal{V}_C$. $\mathcal{V}_S$ is the cut so that all down-streaming layers are processed at the cloud. $\mathcal{V}_E'$ and $\mathcal{V}_S$ are processed at the edge and $\mathcal{V}_C$ are processed at the cloud. We define $\mathcal{V}_E = \mathcal{V}_E' \bigcup \mathcal{V}_S$. The output data of vertices in $\mathcal{V}_S$ will be transmitted from the edge side to the cloud. $\mathcal{V}_E$, including $\mathcal{V}_E'$ and $\mathcal{V}_S$ will generate processing delay at the edge. $\mathcal{V}_S$ will generate transmission delay. $\mathcal{V}_C$ will generate processing delay at the cloud. Our aim is to determine best cut $\mathcal{V}_S$ so that the overall delay is minimized.

As shown in Fig. 6, we cut at $\mathcal{V}_S = \{v_3, v_5, v_9, v_{12}\}$ so that the $\mathcal{V}_E' = \{e, v_1, v_2, v_4\}$, $\mathcal{V}_E = \{e, v_1, v_2, v_3, v_4, v_5, v_9, v_{12}\}$, and $\mathcal{V}_C = \{v_6, v_7, v_8, v_{10}, v_{11}, v_{13}, c\}$. The overall delay is the processing delay of $\mathcal{V}_E$ on the edge and $\mathcal{V}_C$ on the cloud plus the communication delay of the output data of layer in $\mathcal{V}_S$.

*4) Delay Components:* Once the partition is made, each frame is processed at the edge, and then sent from the edge to the cloud, and then processed at the cloud. Since there are multiple frames to be processed, we assume that the three stages are conducted in *pipeline*. In order words, when frame 1 is being processed at the cloud, frame 2 can be transmitted and frame 3 can be processed at the edge.

The delays of the three stages are characterized as follows. In the edge-computing stage

$$T_e = \sum_{v_i \in \mathcal{V}_E} t_i^e. \tag{1}$$

In the cloud-computing stage

$$T_c = \sum_{v_i \in \mathcal{V}_C} t_i^c. \tag{2}$$

In the communication stage

$$T_t = \sum_{v_i \in \mathcal{V}_S} t_i^t. \tag{3}$$

For each frame, $T_e$, $T_c$, and $T_t$ are spent for each stage. Frames are processed in pipeline every $\frac{1}{Q}$. As a consequence, the Gantt chart (scheduling chart) of frames can be shown in Fig. 8. $T_e$, $T_c$, and $T_l$ cannot exceed $\frac{1}{Q}$. Otherwise, the incoming rate is greater than the completion rate, leading to system congestion. **Our aim is to smartly partition the DNN so that the overall delay to process frames is minimized and the system is not congested.**

## C. Parameter Estimation for ECDI

In this subsection, we discuss how to derive the input parameters. The first class of parameters is called DNN profile, including DNN topology $\mathcal{G}$, processing delays of each layer at the edge and the cloud $\mathbf{F}_e$, $\mathbf{F}_c$, data size of each layer $\mathbf{D}_t$. These parameters can be well derived in advance. $\mathcal{G}$ and $\mathbf{D}_t$ can be directly derived given the DNN definition. $\mathbf{F}_e$ and $\mathbf{F}_c$ can be measured beforehand. For example, we derive $\mathbf{D}_t$ of tiny YOLOv2 model and measure $\mathbf{F}_e$ of tiny YOLOv2 model processed on Raspberry Pi 3 model B and Ali Cloud respectively. We show the results in Fig. 1 and Fig. 7 respectively.

The value $B$ is dynamic and should be measured during the process of DNN inference. This can be realized by a method similar to HTTP DASH [11]. We use the tool "ping" at edge to send two different size data consecutively to the cloud, and measure the response times. The bandwidth equals to the ratio between the difference of data size and the difference of response times.

The value $Q$ is user-specific. The user lets the system know $Q$ when the inference starts. The system does nothing unless $Q$ is too large for the system to handle (See Section III-D).

# III. ECDI Partitioning Optimization

## A. The Impact of DNN Inference Workloads

Our first objective is to minimize the overall delay to process each frame. This is true under the *light workload*: for each stage, the current frame is completed before the next frame arrives. Mathematically $\max\{T_e, T_t, T_c\} < \frac{1}{Q}$ so that the Gantt chart is shown as the bottom one of Fig. 8. In this case, we just need to complete every frame *as soon as possible*, i.e., minimize $T_c + T_t + T_e$.

However, if the system is heavily loaded, minimizing $T_e + T_t + T_c$ may lead to system congestion as $\max\{T_e, T_t, T_c\} \geq \frac{1}{Q}$. For example, in Fig. 8 (top), $T_e > \frac{1}{Q}$ so that the next frame arrives before the current frame is completed at the edge. Therefore, under this situation, we need to maximize the throughput of the system, i.e. how many frames at most the system can handle

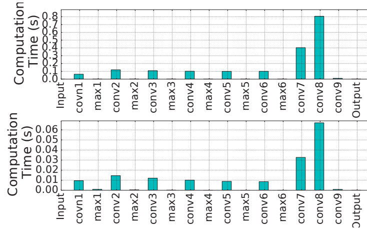


Fig. 7: The computation latency of YOLOv2's layers on the edge (top) and cloud (bottom) respectively.

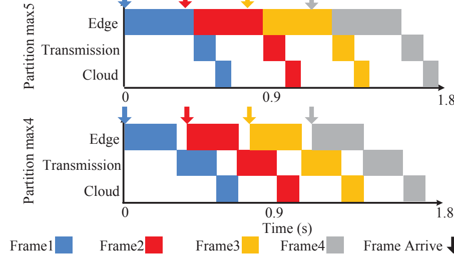


Fig. 8: Gantt charts for three stages.

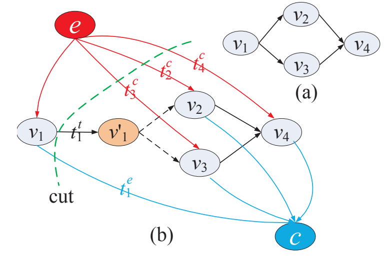


Fig. 9: Illustration of conversion to the minimum s-t cut problem.

per unit time. Our objective is to minimize $\max\{T_e, T_t, T_c\}$ as the system throughput is $\frac{1}{\max\{T_e,T_t,T_c\}}$. For presentation convenience, $\max\{T_e, T_t, T_c\}$ is referred to as the **max stage time**.

Please note that in Section III-D, we will further discuss how to judge if the system is lightly loaded or heavily loaded. There, we also need to consider that if the sampling rate is greater than $\frac{1}{\min\max\{T_e,T_t,T_c\}}$ so that the system will be congested eventually. The system has to force the sender/user to reduce sampling rate.

### *B. The Light Workload Partitioning Algorithm*

In this subsection, we study **Edge Cloud DNN Inference for Light Workload (ECDI-L)** problem. Our goal is to minimize the overall delay of one frame, under a given the network condition $B$. In summary, we have the following optimization problem:

**Problem 1.** *(ECDI-L) Given $\mathcal{G}$, $[\mathbf{F}_e, \mathbf{F}_c, \mathbf{D}_t]$, and $B$, determine $\mathcal{V}_E$, $\mathcal{V}_S$ and $\mathcal{V}_C$, to minimize $T_{inf} = T_e + T_t + T_c$.*

**Proposition 1.** *Problem ECDI-L can be solved in polynomial time.*

One challenge to solve ECDI-L problem directly is that each vertex in $\mathcal{G}$ contains three delay values $t_i^e, t_i^c, t_i^t = \frac{d_i}{B}$. The delay that contributes to the overall delay depends on where the vertex is processed. To this end, we construct a new graph $\mathcal{G}'$ so that each edge only captures *a single* delay value. By doing so, we convert ECDI-L problem to the minimum weighted s-t cut problem of $\mathcal{G}'$.

We first illustrate how to construct $\mathcal{G}'$ based on $\mathcal{G}$.

*a) Cloud Computing Delay:* Based on $\mathcal{G}$, we add links between $e$ and each vertex $v \in \mathcal{V}$, referred to as "red links," to capture the cloud-computing delay of $v$.

*b) Edge Computing Delay:* Similarly, we add links between vertex $v \in \mathcal{V}$ and $c$, referred to as "blue links," to capture the edge-computing delay of $v$.

*c) Communication Delay:* All the other links correspond to communication delays. A link from $v$ to $u$ should capture the communication delay of $v$. However, this is insufficient as one vertex may have multiple successors and its communication delay is counted multiple times. For example, $v_1$ in Fig. 6 has 4 outgoing links but the communication delay of $v_1$ has to be counted at most once. To this end, we introduce *axillary vertices* into graph $\mathcal{G}'$. That is, for any vertex $v_k \in \mathcal{V}$ whose outdegree is greater than one, we add an auxiliary vertex $v_k'$ and link $(v_k, v_k')$. The links from $v_k$ to successors of $v_k$ are now re-placed from $v_k'$ to successors of $v_k$. For example, a 4-layer DNN is shown in Fig. 9(a). The outdegree of vertex $v_1$ is greater than one, we thus add an auxiliary vertex $v_1'$ and link $(v_1, v_1')$ shown in Fig. 9(b). The links $(v_1, v_2)$ and $(v_1, v_3)$ are re-placed by links $(v_1', v_2)$ and $(v_1', v_3)$ respectively. We define $\mathcal{V}_D$ to be the set of axillary vertices.

Now, without considering $e$ and $c$, if a vertex $v$ has one successor, the link starting from $v$ corresponds to its communication delay, which is referred to as "black link." If $v$ has multiple successors, then all the links starting from $v$ are referred to as "dashed links" and should not be considered since the communication delay has already been considered from $v$ to $v'$.

Links are assigned costs. The costs assigned to red, blue, black links are cloud-computing, edge-computing, and communication delays. Dashed links are assigned infinity.

$$c(v_i, v_j) = \begin{cases} t_i^e, & \text{if } v_i \in \mathcal{V}, v_j = c. \\ t_i^t, & \text{if } v_i \in \mathcal{V}, v_j \in \mathcal{V} \bigcup \mathcal{V}_D. \\ t_i^c, & \text{if } v_i = e, v_j \in \mathcal{V}. \\ +\infty, & \text{others.} \end{cases} \tag{4}$$

At this stage, we can convert ECDI-L problem to the minimum weighted s–t cut problem of $\mathcal{G}'$.

A cut is a partition of the vertices of a DAG into two disjoint subsets. The s–t cut of $\mathcal{G}'$ is a cut that requires source $s$ and sink $t$ to be in different subsets, and its *cut-set only consists of links going from the source's side to sink's side*. The value of a cut is defined as the sum of the cost of each link in the cut-set. Problem ECDI-L is equivalent to the minimum $e$–$c$ cut of $\mathcal{G}'$. If cutting on link from $e$ to $v_i \in \mathcal{V}$ (red link shown in Fig. 9(b)), then $v_i$ will be processed on the cloud, i.e $v_i \in \mathcal{V}_C$. If cutting on link from $v_j \in \mathcal{V}$ to $c$ (blue link show in Fig. 9(b)), then $v_j$ will be processed on the edge, i.e. $v_j \in \mathcal{V}_E$. If cutting on link from $v_i \in \mathcal{V}$ to $v_j \in \mathcal{V} \bigcup \mathcal{V}_D$ (black link show in Fig. 9(b)), then the data of $v_i$ will be transmitted to the cloud, i.e $v_i \in \mathcal{V}_S$. It is impossible to cut on link from $v_i \in \mathcal{V}_D$ to $v_j \in \mathcal{V}$ (dashed links), because otherwise it will lead to infinite cost (but finite cost exists). The total cost of cut on red links equals to cloud computation time $T_c$. The total cost of cut on blue links equals to edge computation time $T_e$. The total cost of cut on black links equals to transmission time without network latency $T_t$. If the $e$–$c$ cut of $\mathcal{G}'$ is minimum, then the inference latency on a single frame is minimum. For example, in Fig 9(b), the cut is at $(e, v_2)$, $(e, v_3)$, $(e, v_4)$, $(v_1, v_1')$ and $(v_1, c)$. $v_1$ is processed at the edge so that $t_1^e$ is counted in the blue link. $v_2$, $v_3$ and $v_4$ are processed at the cloud so that $t_2^c$ $t_3^c$ and $t_4^c$ are counted in the

red links. The communication delay $t_1^t$ is counted in the black link.

We develop DNN Surgery Light (denoted as DSL) algorithm for ECDI-L problem. The overall algorithm `DSL()` is shown in Algorithm 1. The algorithm first calls `compute-net()` to compute $\mathbf{F}_t$. Then it calls `graph-construct()` (line 2) to construct $\mathcal{G}'$ based on $\mathcal{G}$ with the computation complexity of $\mathcal{O}(n+m)$, where $n$ is the number of layers $|\mathcal{V}|$, $m$ is the number of links $|\mathcal{L}|$, and then it calls `min-cut()` (line 3) to find minimum $e$–$c$ cut of $\mathcal{G}'$ which outputs the partition strategy (i.e. $\mathcal{V}_E, \mathcal{V}_S$ and $\mathcal{V}_C$). Boykov's algorithm [12] is used in `min-cut()` to solve the minimum $e$–$c$ cut problem with the computational complexity of $\mathcal{O}((m+n)n^2)$. `DSL()` is a polynomial-time algorithm with the computational complexity of $\mathcal{O}((m+n)n^2)$.

**Algorithm 1:** DSL Algorithm `DSL()`.

**Input:** $\mathcal{G}, \mathbf{F}_e, \mathbf{F}_c \mathbf{D}_t, B$
**Output:** $\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_e, T_t, T_c$
1 $\mathbf{F}_t \leftarrow$ `compute-net(`$\mathbf{D}_t, B$`)`;
2 $\mathcal{G}' \leftarrow$ `graph-construct(`$\mathcal{G}, \mathbf{F}_e, \mathbf{F}_c, \mathbf{F}_t$`)`;
3 $[\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_e, T_t, T_c] \leftarrow$ `min-cut(`$\mathcal{G}'$`)`;
4 return $\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_e, T_t, T_c$;

### *C. The Heavy Workload Partitioning Algorithms*

As discussed in Section III-A, we formulate the **Edge Cloud DNN Inference for Heavy Workload (ECDI-H)** problem, to minimize $\max\{T_e, T_t, T_c\}$. The decision variables are $\mathcal{V}_E$, $\mathcal{V}_S$ and $\mathcal{V}_C$. In summary, we have the following optimization problem:

**Problem 2.** *(ECDI-H) Given $\mathcal{G}$, $[\mathbf{F}_e, \mathbf{F}_c, \mathbf{D}_t]$, and $B$, determine $\mathcal{V}_E$, $\mathcal{V}_S$ and $\mathcal{V}_C$, to minimize $\max\{T_e, T_t, T_c\}$ (i.e. maximize throughput).*

ECDI-H Problem is NP-hard. We provide the sketch of the proof. We prove it by reducing from the smallest component of the most balanced minimum st-vertex cut problem (MBMVC-SC), which is known to be NP-complete [13]. We consider the following MBMVC-SC problem on $\mathcal{G}_A = (\mathcal{V}_A \bigcup \{s,t\}, \mathcal{L}_A)$, the goal is to find a vertex cut set $\mathcal{V}_C$ to partition the graph into two disjoint components $(\mathcal{V}_1, \mathcal{V}_2)$, for which $s \in \mathcal{V}_1$, $t \in \mathcal{V}_2$ and the largest components among $\{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_C\}$ is minimum. Any instance of the above problem is equivalent to an instance in ECDI-H problem. Due to page limitation, detailed explanations are omitted.

ECDI-H is NP-hard. It is unrealistic to find a globally optimal solution within polynomial time. We design DNN Surgery Heavy (denoted as DSH) algorithm which achieves a locally optimal solution. In addition, its approximation ratio is 3.

The rationale to develop DSH is as follows. We modify $\mathcal{G}'$ by changing the costs of links as follow:

$$c(v_i, v_j) = \begin{cases} \alpha t_i^e, & \text{if} \quad v_i \in \mathcal{V}, v_j = c. \\ \beta t_i^t, & \text{if} \quad v_i \in \mathcal{V}, v_j \in \mathcal{V} \bigcup \mathcal{V}_D. \\ \gamma t_i^c, & \text{if} \quad v_i = e, v_j \in \mathcal{V}. \\ +\infty, & \text{others.} \end{cases} \tag{5}$$

Here $\alpha, \beta$ and $\gamma$ are non-negative variables. The approach is to run `DSL()` with several different $\alpha, \beta$ and $\gamma$ values. By this way, a solution is generated to optimize ECDI-L with a specific $\alpha, \beta, \gamma$ tuple. Then we test if this solution is also good enough for ECDI-H. If it is better than all existing solutions, it is regarded as a new solution to ECDI-H. We repeat the above procedure for a wide range of $\alpha, \beta, \gamma$ tuples.

Here, the result of `DSL()` is determined by the ratio of the three parameters, instead of their absolute values. Therefore, we can fix one of the three, for example, $\beta = 1$ , and only vary the other two. Thus, we have a two-dimensional search space for $\alpha$ and $\gamma$. We first search in the two-dimensional plane with a coarse granularity to find the best solution. Then we use a finer granularity search in the neighborhood of the best solution for further improvement. We repeat the steps until the improved performance is smaller than a threshold $\epsilon$.

The overall algorithm `DSH()` is shown in Algorithm 2. A function `search()` (line 11–19) is designed to search for the best solution in a given space $\mathbf{S} \triangleq [\alpha_l, \gamma_l, \alpha_h, \gamma_h]$, meaning that $\alpha_l \le \alpha \le \alpha_h, \gamma_l \le \gamma \le \gamma_h$, and a granularity $\delta$ (line 13–14), i.e. the step size of changing $\alpha$ and $\gamma$ is $\delta$ each time. For each $\alpha$ and $\gamma$, `search()` calls `DSL()` to compute the vertex cut and calls `max-time()` to compute the $\max[T_c, T_e, T_t]$. Lines 17–18 guarantee $\max[T_c, T_e, T_t]$ derived is non-increasing.

The overall algorithm first initializes the search granularity $\delta$ to be 1 (line 2) and the search space large enough (line 3–4). It calls `search()` (line 8) to search on the given space $\mathbf{S}$ with a granularity $\delta$, and returns the best $\alpha$ and $\gamma$ found currently. Then `DSH()` narrows down the search space $\mathbf{S}$ (line 8) to the neighborhood of the best $\alpha$ and $\gamma$ for the current iteration, and adjusts $\delta$ to a finer granularity (line 9). Such space $\mathbf{S}$ and granularity $\delta$ is returned to `search()`. The termination condition for the loop is that the improved performance is smaller than a threshold $\epsilon$ (line 5). Finally, it returns the vertex cut with the best-found performance (line 10). Obviously, we can achieve a local optimal result with respect to the neighborhood of the final $\alpha$ and $\gamma$.

**Theorem 1.** *The approximation ratio of the algorithm DSH for ECDI-H is* 3*.*

*Proof.* Let the max stage time of DHL be $t_{DSH}$. Let the optimal max stage time of ECDI-H be $t^*$. We prove $\frac{t_{DSH}}{t^*} \le 3$. Let $T^*$ denote the minimum inference latency for one frame. Let $T_o$ denote the inference latency of a single frame when achieving the optimal max stage time. We have $T^* \le T_o$. Because there are three stages, we have $T_o \le 3t^*$, thus $T^* \le 3t_o$.

As shown in Algorithm 2, when $\delta = 1$, `Search()` will calls `DSL()` using $\alpha = 1$ and $\gamma = 1$ as the parameter. When $\alpha = 1$ and $\gamma = 1$, `DSL()` achieves the minimum inference time $T^*$ for one frame. Let $t_1$, $t_2$ and $t_3$ be the edge computation time, the transmission time and the cloud computation time respectively when achieving the minimum inference time. We have $T^* = t_1 + t_2 + t_3$. `DSH()` guarantees the searched max stage time is non-increasing, thus $t_m \le \max\{t_1, t_2, t_3\}$, combined with $T^* = t_1 + t_2 + t_3$, we have $t_m \le T_{min}$. As $t_m \le T^*$ and $T^* \le 3t^*$, we prove $\frac{t_{DSH}}{t^*} \le 3$. $\square$

**Algorithm 2:** DSH Algorithm `DSH()`.

**Input:** $\mathcal{G}, \mathbf{F}_e, \mathbf{F}_c, \mathbf{D}_t, B, \epsilon, K$
**Output:** $\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_{max}$

1 $\mathbf{F}_t \leftarrow$ `compute-net` $(\mathbf{D}_t, B)$;
2 $T_{max} \leftarrow +\infty$; $T'_{max} \leftarrow 0$; $\delta \leftarrow 1$;
3 $\alpha_l \leftarrow 0$; $\gamma_l \leftarrow 0$; $\alpha_u \leftarrow \frac{\sum(\mathbf{F}_e)}{\min(\mathbf{F}_t)}$; $\gamma_u \leftarrow \frac{\sum(\mathbf{F}_c)}{\min(\mathbf{F}_t)}$;
4 $\mathbf{S} \leftarrow [\alpha_l, \gamma_l, \alpha_u, \gamma_u]$;
5 **while** $|T'_{max} - T_{max}| \geq \epsilon$ **do**
6 $\quad T'_{max} \leftarrow T_{max}$;
7 $\quad [\alpha, \gamma, \mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_{max}] \leftarrow$ `Search` $(\mathbf{S}, \delta, T_{max})$;
8 $\quad \alpha_l \leftarrow \alpha - \delta$; $\alpha_u \leftarrow \alpha + \delta$; $\gamma_l \leftarrow \gamma - \delta$; $\gamma_u \leftarrow \gamma + \delta$;
9 $\quad \delta \leftarrow \delta / K$;
10 return $\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_{max}$;
11 **function** *Search*$([\alpha_l, \gamma_l, \alpha_u, \gamma_u], \delta, T^*_{max})$
12 $\quad T_{max} \leftarrow +\infty$;
13 $\quad$ **for** $\alpha \leftarrow \alpha_l; \alpha \leq \alpha_u; \alpha \leftarrow \alpha + \delta$ **do**
14 $\quad\quad$ **for** $\gamma \leftarrow \gamma_l; \gamma \leq \gamma_u; \gamma \leftarrow \gamma + \delta$ **do**
15 $\quad\quad\quad [\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_e, T_t, T_c] \leftarrow$ `DSL` $(\mathcal{G}, \alpha\mathbf{F}_e, \gamma\mathbf{F}_c, \mathbf{D}_t, B)$
16 $\quad\quad\quad T_{max} \leftarrow$ `max-time` $(T_e, T_t, T_c)$;
17 $\quad\quad\quad$ **if** $T_{max} \leq T^*_{max}$ **then**
18 $\quad\quad\quad\quad \alpha^* \leftarrow \alpha$; $\gamma^* \leftarrow \gamma$; $T^*_{max} \leftarrow T_{max}$;
19 $\quad$ return $\alpha^*, \gamma^*, \mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T^*_{max}$;

### *D. The Dynamic Partitioning Algorithm*

We now consider network dynamics. In practice, the network status $B$ varies. This will affect the workload mode selection and the partition decision dynamically. We design Dynamic Adaptive DNN Surgery (DADS) scheme to adapt network dynamics.

It is shown in Algorithm 3. `monitor-task()` monitors whether the video is active (line 2). This can be realized by tool "iperf." Detailed implementation can be found in Section IV. The real-time network bandwidth is derived by `monitor-net()` (line 3). Then `DSL()` is called to compute the partition strategy (line 4). In this case, if it satisfies the sampling rate $\frac{1}{Q}$, i.e.$\max\{T_e, T_t, T_c\} < \frac{1}{Q}$, we can confirm that the system is in the light workload mode and the partition by DSL is accepted.

Otherwise, the system is in the heavy workload mode and calls `DSH()` to adjust the partition strategy to minimize the max delay (line 6). However, if the completing rate is still smaller than the sampling rate, it means that the sampling rate if too large so that even `DSH()` still cannot satisfy the sampling rate. The system will be congested. It calls the user to decreases the sampling rate (line 7–8).

## IV. Implementation

We implement a DADS prototype system. We use the Raspberry Pi 3 model B as the edge device, integrated with a Logitech BRIO camera. We rent a server in Cloud Ali with 8 cores of 2.5 GHz and a total memory of 128 GB. We employ WiFi as the communication link between the edge device and the cloud. The wired link from the edge router and the cloud is sufficiently large. We implement our client-server interface using GRPC, an open source flexible remote procedure call (RPC) interface for inter-process communication.

**The edge device.** The duty of the edge device is to 1) extract video from the camera and to sample frames from video, 2)

**Algorithm 3:** DADS Algorithm `DADS()`

1 **while** *true* **do**
2 $\quad$ **if** *`monitor-task()`==true* **then**
3 $\quad\quad B \leftarrow$ `monitor-net()`;
4 $\quad\quad [\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_e, T_t, T_c] \leftarrow$ `DSL` $(\mathcal{G}, \mathbf{F}_e, \mathbf{F}_c, \mathbf{D}_t, B)$;
5 $\quad\quad$ **if** $\max\{T_e, T_t, T_c\} > \frac{1}{Q}$ **then**
6 $\quad\quad\quad [\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C, T_{max}] \leftarrow$ `DSH` $(\mathcal{G}, \mathbf{F}_e, \mathbf{F}_c, \mathbf{D}_t, B, \epsilon, K)$;
7 $\quad\quad\quad$ **if** $T_{max} > \frac{1}{Q}$ **then**
8 $\quad\quad\quad\quad$ `inform-decrease()`;
9 $\quad\quad$ `execute` $(\mathcal{V}_E, \mathcal{V}_S, \mathcal{V}_C)$;

make partition decision, 3) process the layers allocated to the edge device, and 4) inform the cloud the partition decision and transfer the intermediate results to the cloud.

For video extraction, we extract videos from camera logitech BRIO using the provided API `video_capture()`. The camera transfers the captured video to Raspberry Pi through the USB-to-serial cable.

For partition decision making, we implement a process that monitors the generated frame by the camera, and runs DADS scheme. DADS requires to estimate the real-time network bandwidth. We use the command "iperf" provided by the operation system Raspbian on Raspberry Pi. This command feeds back the real-time network bandwidth between the Raspberry Pi and the cloud.

For processing allocated layers on the edge, we install a modified instance of Caffe and store a full DNN model on the edge device. The challenge is to control Caffe to stop execution at partitioned layers (e.g., $\mathcal{V}_S$). In Caffe, there is a "prototxt" file recording the DNN structure. Layers are processed according to this file. To solve the challenge, we modify the model structure file "prototxt" by inserting a "stop layer" after each partitioned layer. The instance of Caffe will stop processing at the desired places.

For the intermediate results and partition decision transmission, the edge device calls the RPC function `receiveRPC()` provided by the cloud to transmit the data to the cloud.

**The cloud.** The duty of the cloud is to execute the DNN layers allocated to the cloud. There are two jobs: 1) to receive the partition decision and the intermediate results from the edge device, and 2) to execute the layers allocated to the cloud.

For the first job, we expose an API `receiveRPC()` to the edge device. After completing processing layers allocated to the edge, the edge device calls this RPC function to transmit the intermediate results packed with the partition decision to the cloud.

For the second job, we implement a modified instance of Caffe and store a full DNN model. The challenge is to execute only the layers allocated to the cloud. To this end, after receiving the partition decision and intermediate results, the layers allocated to the edge are deleted before the marked place in "prototxt," and the intermediate results are forwarded to the corresponding layers as input. By this way, only layers allocated to the cloud will be executed.

## V. Performance Evaluation

We evaluate the DADS prototype (Section IV) using real-trace driven simulations.

### A. Setup

**Video Datasets.** We employ the publicly available BDD100K self-driving dataset. The videos of this dataset are obtained from the camera on the self-driving car. Each video is about 40 seconds long and is viewed in 720p at 30 FPS.

**Workload Setting.** We divide the inference task into low workload mode and heavy workload mode. Accordingly, We transform the video into different sampling rates to produce different workload. We set a low sampling rate to 0.1 frame per second when evaluating light workload mode, and 20 frames per second for heavy workload mode. The default resolution is 224p. Each inference task consists of processing 100 frames using the given DNN benchmarks.

**Communication Network Parameters.** To model the communication between edge and cloud, we used the average uplink rate of mobile Internet for different wireless networks, i.e. CAT1, 3G, 4G and WiFi as shown in Table I.

**DNN Benchmarks.** DADS can make partition not only on chain topology DNN but also on the DAG topology. We evaluate the performance of DADS for both topologies. For the chain topology, NiN, tiny YOlOv2 and VGG16, are well-known models used as benchmarks in this evaluation shown in Fig. 10. For the DAG topology, we employ AlexNet and ResNet-18 as the benchmarks shown in Fig. 11.

**Evaluation Criteria:** We compare DADS against Edge-Only (i.e. executing the entire DNN on the edge), Cloud-Only (i.e. executing the entire DNN on the cloud), and a variant Neurosurgeon which is a partition strategy for chain-topology DNN. To evaluation Neurosurgeon's performance for DAG, we consider a variant Neurosurgeon, which first employs topological sorting method to transform the DAG topology to the chain topology, and then uses the original partition method. We use the Edge-Only method as the baseline, i.e. the performance is normalized to Edge-Only method.

We evaluate the latency and throughput of DADS compared with Edge-Only, Cloud-Only and Neurosurgeon in Section V-B. We also evaluate the impact of different types of wireless network to DADS, and the impact of bandwidth on the selection of workload mode in Section V-C.

### B. Performance Comparison

We first compare our DADS with Edge-Only, Cloud-Only and Neurosurgeon under light workload mode and heavy workload mode across the 5 DNN benchmarks in Fig. 12–14. The results are normalized to Edge-Only method. We see that DADS achieves a higher latency speedup and throughput gain compared with other methods.

**Comparing DADS with Edge-Only and Cloud-Only:** DADS has a latency speedup of 1.91–6.45 times, 1.35–8.08 times compared with Edge-Only and Cloud-Only methods respectively under the light workload mode shown in the bottom graph of Fig. 12. DADS has a throughput gain of 3.45–8.31 times, 1.46–11.13 times compared with Edge-Only and Cloud-Only methods respectively under the light workload mode shown in the upper graph of Fig. 12. This is because, Edge-Only method executes the entire DNN on the edge side, it avoids data transmission and ignores the weak computation capacity of edge side. Cloud-Only method ignores the effect of the transmission time. DADS considers both computation and transmission, and it makes a good tradeoff between them.

From Fig. 16, we can see that, for the heavy workload mode, DADS outperforms Edge-Only and Cloud-Only 1.66–5.19 times and 1.07–6.92 times respectively in latency reduction, and DADS outperforms Edge-Only and Cloud-Only 4.34–9.14 times and 1.46–14.10 times respectively in throughput gain. This further confirms that DADS significantly outperforms Edge-Only and Cloud-Only methods.

**Comparing DADS with Neurosurgeon:** Neurosurgeon can automatically partition DNN between the edge device and cloud at granularity of neural network layers, but it is only effective for chain topology.

From Fig. 14, we can see that, for the chain topology models, DADS and Neurosurgeon have the similar performance in latency and throughput for the light workload. While for the heavy workload, Neurosurgeon has a latency reduction of 16.28% and 13.64% than that of DADS for YOLOv2 and VGG16, however the throughput gain of DADS is 1.26 times and 1.27 times than that of Neurosurgeon under these two DNN models. This is because, for the heavy workload, the higher throughput is prior for DADS. We also can see that, for the heavy workload and NiN model, the latency and the throughput of Neurosurgeon and DADS are both the same. This is because for NiN model, DADS achieves the minimum max stage time when the latency is minimum.

For the DAG topology, we can observe that DADS outperforms Neurosurgeon significantly. For DAG topology models, DADS has a latency speedup 66%–86% and throughput gain of 76%–87% compared with Neurosurgeon. This observation validates the usefulness of DADS for DAG topology.

### C. Network Variation

In this section, we evaluate how transmission network affects the performance of DADS using ResNet18 model. The sampling rate is 1 frame per second.

**The Impact of Transmission Network Type:** We first evaluate the performance of DADS, Edge-Only and Cloud-Only for ResNet18 model when using Cat1, 3G, 4G and WiFi as the communication network.

In Figs. 15–16, we show the latency speedup and the throughput gain achieved by DADS and Cloud-Only normalized to Edge-Only when using Cat1, 3G, 4G and WiFi for light and heavy workload respectively.

Shown in Fig. 15, when the workload is light and the edge device communicates with the cloud through Cat1, DADS achieves 1.46 times latency reduction and 2.03 times throughput gain compared with Edge-Only. When the network changes to 3G, 4G and 5G the latency reduction and the throughput gain becomes more significant: 4.14 times and 8.3 times for 3G, 7.23 times and 9.78 times for 4G, 8.32 times and 9.31 times for

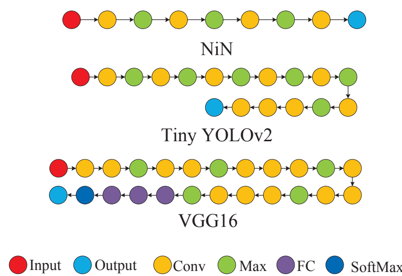


Fig. 10: The chain-topology DNN models.

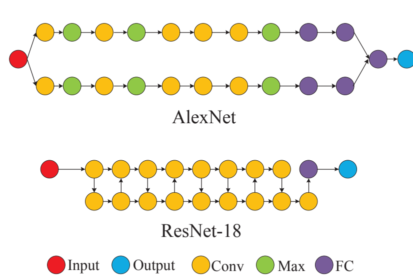


Fig. 11: The DAG-topology DNN models.

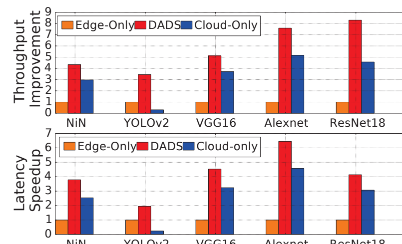


Fig. 12: Latency speedup and throughput gain achieved by DADS under light workload mode.

TABLE I: DNN Benchmark Specifications

| | CAT1 | 3G | 4G | WiFi |
|---|---|---|---|---|
| Uplink rate (Mpbs) | 0.13 | 1.1 | 5.85 | 18.88 |

TABLE II: DNN Benchmark Specifications

| Type | Chain | | | DAG | |
|---|---|---|---|---|---|
| Model | NiN | YOLOv2 | VGG16 | Alexnet | ResNet18 |
| Layers | 9 | 17 | 24 | 23 | 20 |

WiFi respectively. When the communication link provides more bandwidth, DADS pushes larger portions of layers to the cloud to achieve better performance. We can also see that, compared with Cloud-Only, DADS achieves latency reduction of 64% for CAT1, 26% for 3G and 7% for 4G receptively, and throughput gain of 73% for CAT1, 45% for 3G and 4% for 4G. For WiFi, the performance of Cloud-Only is good enough, it has the same performance with DADS.

Edge-Only is only good for low data rate. Cloud-Only is only good for high data rate, DADS can be adaptive to a wide range of network setting.

**The Impact of Bandwidth on Workload Mode Selection:** In Fig. 17, we show the workload mode switch of DADS under different network bandwidth. We can see that when the available bandwidth is smaller than 1.51Mbps, DADS works at heavy workload mode, and the achieved latency speedup and throughput gain increase compared with Edge-Only. When the bandwidth is greater than 1.51Mbps, DADS works at light workload mode.

We also evaluate DADS's resilience to real-world measured wireless network variations. In Fig. 18, the top graph shows measured wireless bandwidth over a period of time. The bottom graph shows the latency speedup of DADS normalized to Edge-Only for ResNet18 model. We can see that DADS adjusts the partition strategy according to the bandwidth variance successfully. For example, when the bandwidth drops from 3.41Mbps to 2.15Mbps, DADS changes the partition from conv2 layer to conv3 layer. DADS changes the partition from conv3 layer to conv7 layer when bandwidth is smaller than 1.72Mbps.

## VI. Related Work

**Modification of DNN Models.** In order to realize inference acceleration, one category of related work investigated how to modify DNN models for speedup. For example, Microsoft and Google developed small-scale DNNs for speech recognition on mobile platforms by sacrificing the high prediction accuracy [14]. MCDNN [15] proposed generating alternative DNN models to trade off accuracy and performance/energy and choosing to execute either in the cloud or on the mobile. [16] proposed deep models that are much smaller than normal and to be run on phones. [17] allowed to use a pool of DNNs and the most effective one is selected to use at runtime. Our proposed DADS does not modify DNNs. It employs full-scale deep models without sacrificing accuracy.

**Computation Offloading.** Research efforts focusing on offloading computation from the resource-constrained mobile to the powerful cloud will reduce inference time. Neurosurgeon [18] explored a computation offloading method for DNNs between the mobile device and the cloud server at layer granularity. However, Neurosurgeon is not applicable for the computation partition performed by DADS for a number of reasons: 1) Neurosurgeon only handle chain-topology DNNs that are much easier to process. 2) Neurosurgeon can only handle one inference task, without considering a sequence of tasks. Needless to say the adaptation to network condition realized by DADS. MAUI's [19] is an offloading framework that can determine where to execute functions (edge or cloud) of a program. However, it is not designed specifically for DNN partitioning as the communication data volume between functions is small. [20] proposed DDNN, a distributed deep neural network architecture that is distributed across computing hierarchies, consisting of the cloud, the edge and end devices. DDNN aims at reducing the communication data size among devices for the given DNN. DADS differs as it handles dynamic network condition to reduce the inference latency (communication and computing latency) rather than communication overhead only.

**Hardware Acceleration.** Different from the scope of this paper. Hardware specialization is another method for inference acceleration. [21] proposed DeepBurning, an automation tool to generate FPGA-based accelerators for DNN models. Vanhoucke et al. [22] used fixed point arithmetic and SSSE3/SSE4 instructions on x86 machines to reduce the inference latency. DeepX [23] explored the opportunities to use mobile GPUs to enable real-time deep learning inferences. DADS investigates intelligent collaboration between the edge device and cloud for inference optimization and can be jointly applied with specialized hardware.

## VII. Conclusion

In this paper, we study DNN inference acceleration through collaborative edge-cloud computation. We propose Dynamic Adaptive DNN surgery (DADS) scheme that can partition

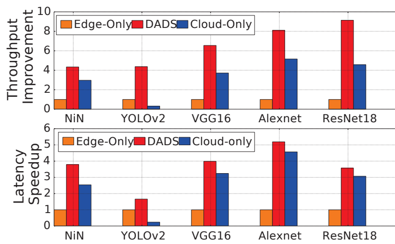


Fig. 13: Latency speedup and throughput gain achieved by DADS under heavy workload mode.

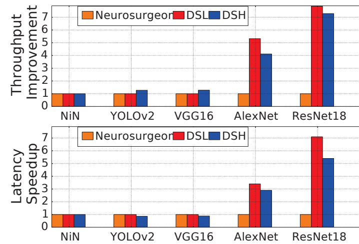


Fig. 14: Latency and throughput speedup achieved by DADS vs. Neurosurgeon under light and heavy workload modes.

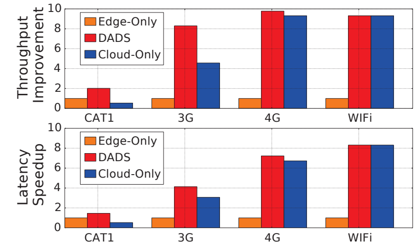


Fig. 15: Latency speedup and throughput gain achieved by DADS of different networks under light workload mode.

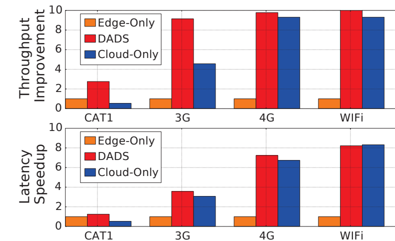


Fig. 16: Latency speedup and throughput gain achieved by DADS of different networks under heavy workload mode.

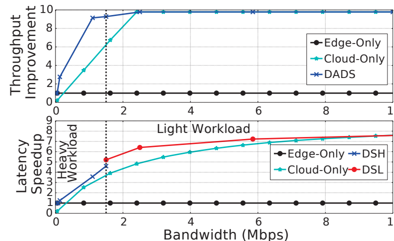


Fig. 17: Latency speedup and throughput gain achieved by DADS as a function of bandwidth.

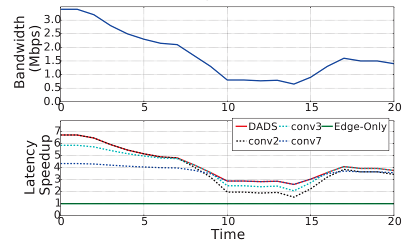


Fig. 18: The impact of network variance on DADS partition decision using Edge-Only as the baseline.

DNN inference between the edge device and the cloud at the granularity of neural network layers, according to the dynamic network status. We present a comprehensive study of the partition problem under the lightly loaded condition and the heavily loaded condition. We also develop an optimal solution to the lightly loaded condition by converting it to min-cut problem, and design a 3-approximation ratio algorithm under the heavily loaded condition as the problem is NP-hard. We then implement a fully functioning system. Evaluations show that DADS can effectively improve latency and throughput in an order compared with executing the entire DNN on the edge or on the cloud.

## VIII. Acknowledgments

We would like to acknowledge the support of the Hong Kong Polytechnic University under Grant PolyU G-YBQE and Innovation Technology Fund (ITF)-UICP-MGJR UIM/363. This work was also supported by the University of Sydney DVC Research/Bridging Support Grant.

## References

[1] V. Va, T. Shimizu, G. Bansal, R. W. Heath Jr *et al.*, "Millimeter wave vehicular communications: A survey," *in Now Publishers Journal on Foundations and Trends in Networking*, vol. 10, no. 1, pp. 1–113, 2016.

[2] "State of Mobile Networks: USA." https://opensignal.com/reports/2017/08/usa/state-of-the-mobile-network, accessed June, 2018.

[3] "AWS DeepLens," https://aws.amazon.com/deeplens, accessed June, 2018.

[4] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[5] D. Raca, J. J. Quinlan, A. H. Zahran, and C. J. Sreenan, "Beyond throughput: a 4G LTE dataset with channel and context metrics," in *Proc. ACM MMSys'18*, Amsterdam, The Netherlands, Jun. 2018.

[6] M. Franceschinis, M. Mellia, M. Meo, and M. Munafo, "Measuring TCP over WiFi: A real case," in *1st workshop on Wireless Network Measurements*, Riva Del Garda, Italy, Sep. 2005.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE CVPR'15*, Boston, MA, Jun. 2015.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR'16*, Las Vegas, Nevada, Jul. 2016.

[9] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live Video Analytics at Scale with Approximation and Delay-Tolerance." in *Proc. USENIX NSDI'17*, Boston, MA.

[10] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *Proc. AAAI'17*, San Francisco, CA, Feb. 2017.

[11] T. Stockhammer, "Dynamic adaptive streaming over HTTP: standards and design principles," in *Proc. ACM MMSYS '11*, Santa Clara, CA, Feb. 2011.

[12] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1124–1137, 2004.

[13] P. Bonsma, "Most balanced minimum cuts," *Discrete Applied Mathematics*, vol. 158, no. 4, pp. 261–276, 2010.

[14] P. Aleksic, M. Ghodsi, A. Michaely, C. Allauzen, B. Hall, D. Rybach, and P. Moreno, "Bringing contextual information to google speech recognition," in *Proc. INTERSPEECH'15*, Dresden, Germany, Sep. 2015.

[15] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *Proc. ACM MobiSys'16*, Singapore, Jun. 2016.

[16] E. Variani, X. Lei, E. McDermott, I. L. Moreno, and J. Gonzalez-Dominguez, "Deep neural networks for small footprint text-dependent speaker verification," in *Proc. IEEE ICASSP'14*, Florence, Italy, May 2014.

[17] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive selection of deep learning models on embedded systems," *arXiv preprint arXiv:1805.04252*, 2018.

[18] 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 *Proc. ACM ASPLOS'17*, Xi'an, China, Apr. 2017.

[19] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proc. ACM MobiSys'10*, San Francisco, CA, Jun. 2010.

[20] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE ICDCS'17*, Atlanta, GA, Jun. 2017.

[21] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *Proc. ACM DAC'16*, Austin, TX, Jun. 2016.

[22] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. NIPS'11*, Granada, Spain, Jan. 2011.

[23] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. IEEE IPSN'16*, Vienna, Austria, Apr. 2016.