# BPS: Batching, Pipelining, Surgeon of Continuous Deep Inference on Collaborative Edge Intelligence

Xueyu Hou, Yongjie Guan, Nakjung Choi, *Senior Member, IEEE*, and Tao Han, *Senior Member, IEEE*

***Abstract*—Users on edge generate deep inference requests continuously over time. Mobile/edge devices located near users can undertake the computation of inference locally for users, e.g., the embedded edge device on an autonomous vehicle. Due to limited computing resources on one mobile/edge device, it may be challenging to process the inference requests from users with high throughput. An attractive solution is to (partially) offload the computation to a remote device in the network. In this paper, we examine the existing inference execution solutions across local and remote devices and propose an adaptive scheduler, a BPS scheduler, for continuous deep inference on collaborative edge intelligence. By leveraging data parallel, neurosurgeon, reinforcement learning techniques, BPS can boost the overall inference performance by up to $8.2\times$ over the baseline schedulers. A lightweight compressor, FF, specialized in compressing intermediate output data for neurosurgeon, is proposed and integrated into the BPS scheduler. FF exploits the operating character of convolutional layers and utilizes efficient approximation algorithms. Compared to existing compression methods, FF achieves up to 86.9% lower accuracy loss and up to 83.6% lower latency overhead.**

***Index Terms*—Edge computing, efficient AI, reinforcement learning, convolutional neural networks.**

## I. Introduction

Convolutional neural networks (CNNs) have achieved remarkable success in computer vision tasks due to their high accuracy [1], [2], [3]. However, they are often characterized by high computational demands, necessitating a large number of operations [4], [5]. To address this, various compression techniques have been developed, aiming to maintain high predictive accuracy while reducing computational costs through modifications to the original CNN architectures [6], [7], [8], [9], [10], [11]. Despite their effectiveness, these methods typically require additional retraining post-compression to regain accuracy, significantly increasing both cost and design overhead [12]. As an alternative, several studies have explored the distribution of CNN model execution across multiple edge/mobile devices to expedite inference [5], [9], [12], [13], [14], [15]. A notable approach, Neurosurgeon [5], proposes reducing computational load on edge/mobile devices by offloading part of a CNN model to a high-performance edge/cloud server. This technique involves dividing the CNN model at an intermediate layer, where the initial layers are computed locally on the edge/mobile device, and the remaining layers are processed on the server, with intermediate outputs transmitted over the network. Subsequent research has expanded on this concept, focusing on areas like intermediate data compression [15], video analytics [16], and integrating model compression techniques [14], among others.

In practical scenarios, CNN model inference requests are continuously generated by users [17], [18], [19], leading to input queues on devices. While local execution of all inferences is feasible, leveraging the computing resources of nearby mobile/edge devices can distribute the computational load. Our findings, which we detail in Section III-A, reveal that combining data parallelism with the Neurosurgeon approach yields the highest inference throughput, effectively minimizing latency across local and remote devices.

In this paper, we address the pressing issue of performing deep inference for multiple users when only one local edge device and another remote edge device are available. This challenge is increasingly relevant in a variety of applications. For instance, in autonomous driving, a local edge device installed in the vehicle processes real-time data from multiple cameras. The nearby infrastructure can act as the remote edge device, assisting in the deep inference. In the realm of video surveillance such as hospitals or residential areas, local edge devices can collect and process video feeds from various rooms. Here, a guardian's device or monitoring system can serve as the remote edge device. For multi-user augmented reality experiences, such as interactive sessions in a conference room, a local computer can process the immediate data from AR glasses, while a more distant edge device might also be engaged for the data processing. These examples illustrate the growing necessity for deep inference in compact spaces like homes, conference rooms, or neighborhood blocks. With the advancement of edge computing, local edge devices are increasingly available in such areas, providing nearby users with network connectivity and computational resources. In a broader public network context, another edge device, reachable via remote network connections, can offer additional computational power for deep inference tasks.

A key distinction of our work is that we do not assume the need for a high-power server as the remote device. Instead, we consider scenarios where the remote device may have a wide range of computing capabilities. This approach broadens

Manuscript received 23 October 2023; revised 14 February 2024; accepted 20 April 2024. Date of publication 10 May 2024; date of current version 6 September 2024. The work of Xueyu Hou, Yongjie Guan, and Tao Han was supported by the National Science Foundation under Grant 2147623 and Grant 2147624. Recommended for acceptance by J. Zhai. *(Corresponding author: Tao Han.)*

Xueyu Hou, Yongjie Guan, and Tao Han are with the New Jersey Institute of Technology, Newark, NJ 07102 USA (e-mail: xh29@njit.edu; yg274@njit.edu; tao.han@njit.edu).

Nakjung Choi is with the Nokia Bell Labs, Murray Hill, NJ 07974 USA (e-mail: nakjung.choi@nokia-bell-labs.com).

Digital Object Identifier 10.1109/TCC.2024.3399616

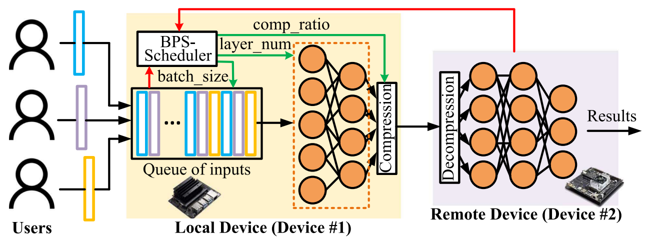

Fig. 1. System overview.

the applicability of our research, making it relevant to a wider array of scenarios, especially where high-performance servers are not feasible or available. By focusing on optimizing the use of remote devices with comparable capabilities to local devices, our paper aligns with the trend of upgrading computing power in mobile and edge devices, making it highly relevant in the current technological landscape.

In our system architecture, depicted in Fig. 1, our primary goal is to enhance the performance of continuous deep inference across two collaborative edge devices. The local device consistently receives user-generated inference requests, while a connected remote device, equipped with computational capabilities, assists in handling part of these computational demands. Although a basic integration of data parallelism and the Neurosurgeon approach already surpasses existing methods in this collaborative scenario, we achieve further performance improvements by implementing two advanced techniques: FastFiltering (FF) compression (see Section IV) and Batch-Pipeline-Surgeon (BPS) scheduling (see Section V). As detailed in Section III-B, the synergistic application of BPS and FF boosts system performance by as much as 44.6%. FF, in particular, stands out as an effective compression tool for intermediate CNN model outputs. It leverages the inherent characteristics of convolutional layers to efficiently eliminate redundant information from the original intermediate data. FF not only maintains a high compression rate but also minimizes overhead. As demonstrated in Section IV, FF achieves up to 18.9% lower accuracy loss and 83.6% less latency overhead compared to other state-of-the-art (SOTA) intermediate data compression techniques [15], [20], [21]. BPS, on the other hand, is a scheduler that optimizes the batch size, cutting layer, and compression parameters for collaborative execution. It dynamically adjusts scheduling decisions based on system condition changes, utilizing reinforcement learning (RL) techniques to ensure high performance (i.e., high accuracy and low latency) under various conditions. As we will show in Section V, the combined use of BPS and FF significantly outperforms baseline scheduling methods, enhancing performance by up to $2.4\times$. The contributions of this paper are as follows:

- Our work introduces FastFiltering (FF), an innovative compression technique specifically designed for the intermediate output data of CNN models. FF leverages the unique operational characteristics of convolutional layers, coupled with efficient lightweight approximation algorithms. This approach enables FF to surpass existing state-of-the-art (SOTA) compression methods, achieving up to 86.9% reduction in accuracy loss and up to 83.6% decrease in latency overhead.
- We have developed an innovative scheduler tailored for continuous deep inference in collaborative edge intelligence environments, named Batch-Pipeline-Surgeon (BPS). Utilizing the capabilities of reinforcement learning, BPS dynamically adapts to changing system conditions in real-time.
- We incorporate the FF compression technique into our BPS scheduler. This integration of BPS with FF significantly enhances the system's overall performance. Specifically, it results in an improvement of up to $49.6\times$ compared to solely local inference, up to $19.2\times$ compared to inference solely relying on offloading, and up to $2.4\times$ when compared to data-parallel inference methods.
- Our evaluation covers a comprehensive range of scenarios. This includes tests on four different pairs of edge devices, each with varying levels of computing capabilities. Additionally, we assess their performance across seven distinct CNN models and under conditions where background applications are running on the devices. For a thorough comparison, FF is benchmarked against two leading state-of-the-art compression methods, while BPS is evaluated alongside six established baseline schedulers.

## II. System Overview

As shown in Fig. 1, we observe a scenario where multiple users submit deep learning inference requests to a local device (Device #1). Concurrently, a remote device (Device #2) is also available to assist with model inference. The incoming requests are queued on Device #1, awaiting processing. To ensure the versatility and applicability of our system, we address several key considerations: *First*, there is **no** inherent correlation between the requests of different users. Unlike systems that leverage spatial similarities among nearby cameras [22], our focus is on a more generalized Machine Learning as a Service (MLaaS) framework [1], [2], [3]. Here, users may send unrelated inputs (images) for distinct deep inferences, catering to a diverse range of requests. *Second*, there is **no** assumed correlation between consecutive inputs from the same user. This perspective diverges from existing approaches that exploit frame-to-frame similarity in video streams [23], [24], [25]. In such works, key frames are selectively processed by CNN models, with other frames being analyzed based on cross-frame similarities, such as tracking algorithms [23]. Thus, in our system, the inputs requiring deep inference are those unique or key frames [16] that demand full model processing. *Third*, the role of the remote device (Device #2) is not limited to high-performance computing resources like cloud or edge servers [5], [15]. Instead, it can be any edge device available within the network, broadening the scope and flexibility of our system to accommodate various computational capabilities.

## III. Preliminary Study

### A. Existing Inference Execution Methods

Consider a device receiving $N$ inputs (e.g., images) from users, and another device available over a network. We can process these inputs using a CNN model $\mathcal{M}$ in several ways:

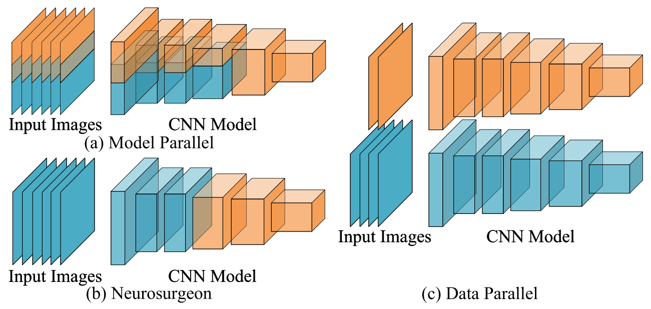

Fig. 2. Demonstration of execution methods across local device (orange) and remote device (blue).

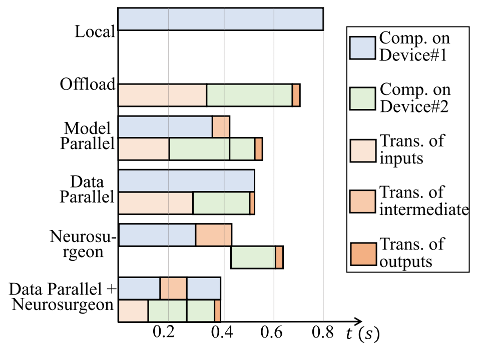

Fig. 3. Comparison of different execution methods.

*1) Local:* We can process all the inputs on the local device. The processing latency is $T = T_{comp}(N, b)$, where $b$ is inference batch size. *(2) Offload:* We can also offload all the inputs to the other device in the network. The processing latency is $T = T_{tran}(N, b, \theta) + T_{comp}(N, b)$, where $\theta$ is network throughput. *(3) Model Parallel:* We can split the first several layers in the original model $\mathcal{M}$ into two parts [26], and run these two parts in parallel on the two devices. The rest layers of $\mathcal{M}$ are executed on one of the two devices. As demonstrated in Fig. 2(a), the first four layers of the CNN model are split into two parts for model paralleling execution. The processing latency is $T = T_{para}(N, b) + T_{comp}(N, b)$. *(4) Data Parallel:* We can also process $N_1$ inputs locally and $N_2$ inputs on the other device ($N = N_1 + N_2$). As demonstrated in Fig. 2(c), we divide the input images into two groups and process them on the local and remote device, respectively. The total processing latency is $T = \max T_{comp1}(N_1, b_1), T_{tran}(N_2, b_2) + T_{comp2}(N_2, b_2)$. *(5) Neurosurgeon:* We can cut the original model $\mathcal{M}$ at an intermediate layer [5]. The first part (first $L_1$ layers) is executed on the local device, and the second part (last $L_2$ layers) is executed on the other device. As demonstrated in Fig. 2(b), we cut the CNN model at the forth layer and execute the two parts on the local and remote device, respectively. The processing latency is $T = T_{comp}(N, b, L_1) + T_{tran}(N, b, o_{L_1}) + T_{comp}(N, b, L_2)$, where $o_{L_1}$ is the intermediate output from layer $L_1$.

In our setups, we measure the latency components involved in the computing and data transmission process between two devices as follows. Specifically, the computing latency on Device #1 is defined as the duration from the moment Device #1 loads all the input data for inference to when it generates the output or intermediate data. Conversely, the computing latency on Device #2 encompasses the time interval from loading the input or intermediate data for inference to producing the final output data. The transmission latency for inputs is determined by the time elapsed from when Device #1 sends the input data to the point at which Device #2 receives them. Similarly, the transmission latency for intermediate data is calculated from when Device #1 transmits these data until Device #2 receives them. Lastly, the transmission latency for outputs is assessed from the moment Device #2 sends the output data to when they are received by Device #1. To ensure accuracy and reliability in our latency measurements, each component is repeatedly tested 50 times under identical test conditions. We then calculate the average of these measurements to determine the final latency values that are reported in the paper.

As shown in Fig. 3, we implement the above five methods in a real setup. The setup includes a Jetson Nano as the local device (Device #1), a Jetson TX2 as the remote device (Device #2), and the network connection between the two devices is WiFi 2.4 GHz. The local device has 20 images, and a ResNet-50 model is utilized to process the images. For each method, we vary batch size $b$ (or $b_1$, $b_2$) to exhaustively find the one(s) that generate the lowest processing latency. For Neurosurgeon, we also search for the optimal intermediate layer exhaustively like [5]. Thus, the processing latency shown in Fig. 3 of each method is their lowest value with the method. Parallel execution methods on both devices, namely model parallel and data parallel, generally outperform other approaches by fully utilizing computing resources and avoiding idle states. The local, offload, and neurosurgeon methods show larger processing latencies due to their inability to simultaneously engage both devices. Additionally, Neurosurgeon benefits from reduced transmission latency, which allows for a more significant computational load on the remote device, as highlighted in Neurosurgeon's approach [5]. This understanding led us to the data parallel + neurosurgeon method, which optimally uses both devices' resources and adaptively manages transmission latency. This approach, as shown in Fig. 3, reduces processing latency by 27.3% to 50% compared to existing methods. We prefer data parallel over model parallel due to the latter's complexity and potential for increased latency from redundant computations [27]. Therefore, our focus is on the synergy of data parallel processing with Neurosurgeon for effective continuous deep inference optimization.

## B. Challenges

Though data parallel + neurosurgeon can be implemented and achieves significant latency reduction in our preliminary setup, a general platform with multiple users sending deep inference requests continuously (Fig. 1) presents challenges:

*First*, while the data parallel + neurosurgeon approach improves processing efficiency, it still faces significant overhead due to intermediate data compression. To evaluate different compression methods, as depicted in Fig. 4, we conduct tests on four device pairs, comprising Jetson Nano and Raspberry Pi4 as local devices, and Jetson TX2 and Jetson Xavier as remote devices,

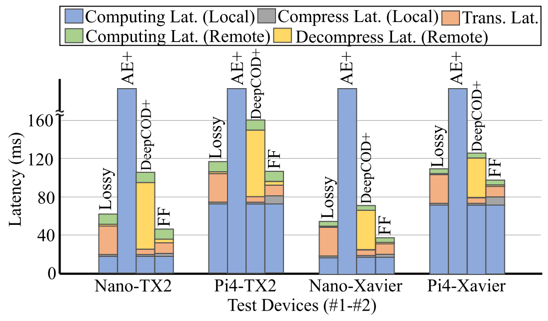

Fig. 4. Comparison of different compression methods (VGG-16, WiFi 2.4 GHz).

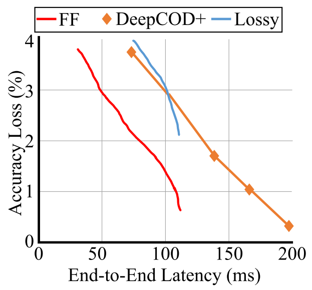

Fig. 5. Comparison of different compression methods (accuracy loss versus end-to-end latency). VGG-16, WiFi 2.4 GHz, Nano-TX2.

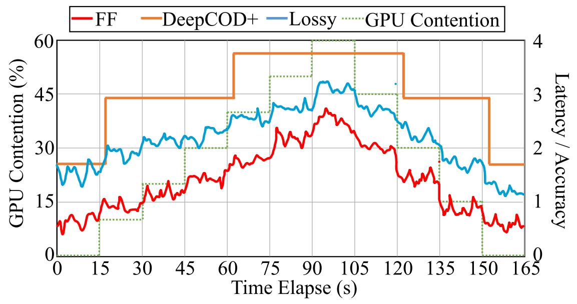

Fig. 6. Performance under changing GPU contention on remote device.

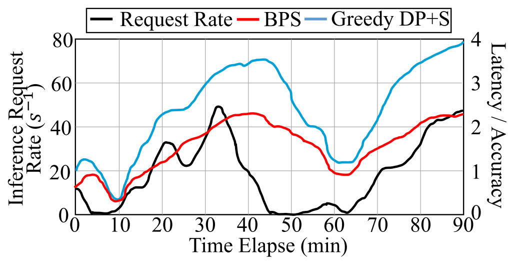

Fig. 7. Dynamic trend of inference request rate.

connected via WiFi 2.4 GHz. Using the VGG-16 model (batch size of 4) and maintaining a consistent cutting layer (Conv 3.1 in VGG-16), we compared latencies for each method, ensuring less than 4% latency loss. We evaluated the *Lossy* method, choosing between JPEG compression and Huffman Coding for optimal compression rates [20], [21]. The *AE+* method involved using encoder-decoder neural networks [28] compressed with a state-of-the-art model compression technique [29]. The *DeepCOD* method [15] was also assessed with a pruned decoder (termed DeepCOD+), leading to additional accuracy loss but reduced latency. Despite these modifications, DeepCOD+ exhibited considerable decompression delays across all device combinations.

*Second,* for an effective compression method, maintaining low end-to-end latency without significantly impacting accuracy is crucial. As illustrated in Fig. 5, lossy compression methods can achieve higher compression ratios at the cost of accuracy. For instance, in a Nano-TX2 setup with WiFi 2.4 GHz, we observe an accuracy loss of 3.94% for an end-to-end latency of 75.4 ms, though higher accuracy loss can lead to even lower latencies. DeepCOD+, despite model compression, still suffers from high decompression latency, constituting 48.6% to 81.3% of total latency and resulting in a 3.78% accuracy loss for a 72.8 ms latency. In contrast, our FF method exemplifies Pareto optimality, achieving just a 1.01% accuracy loss at only 60.1% of DeepCOD+'s latency; for similar accuracy losses, FF's latency is merely 50.6% to 60.3% of that incurred by lossy methods. Moreover, smooth accuracy-latency trade-off curves enhance adaptability in dynamic conditions. As shown in Fig. 6, under changing GPU contention, DeepCOD+ shows significant fluctuations, reflecting its poorer adaptability compared to FF.

*Third,* the real-world applications involve continuous input arrival at the local device, leading to fluctuating numbers of inputs awaiting processing. This variability, influenced by the unpredictable nature of user behavior [18], results in a widely fluctuating inference request rate. For instance, as depicted in Fig. 7, the request rate from four street cameras (sending only keyframes for tiny-YOLOv3 object detection) varied between 0 to 49 frames per second over 90 minutes. A greedy data parallel + neurosurgeon scheduler (Greedy DP + S) processes the current queue without considering incoming request rate trends, leading to suboptimal scheduling. In contrast, our proposed BPS scheduler employs reinforcement learning to anticipate future system states based on current conditions, including network throughput, device contention, and queue length. Fig. 7 demonstrates that BPS consistently outperforms the greedy approach, especially during periods of high request rate variability (10 to 45 minutes and 65.5 to 90 minutes), achieving up to 44.6% ratio reduction in latency over accuracy.

## IV. FastFiltering Compression

Driven by the limitations of existing data compression techniques outlined in Section III-B, we introduce FastFiltering (FF), a lightweight method for compressing intermediate outputs in CNN models. FF efficiently identifies and removes redundant information across a model's feature maps, both in depth and spatial dimensions. This process significantly reduces the transmission data size, as only essential information is sent to the remote device. FF's low latency and instant online tuning capabilities allow for adaptive performance in varying network and device conditions. The FF workflow, shown in Fig. 8, comprises three main modules: Depth Element Selection (DES) and Spatial Vector Clustering (SVC) on the local device, and Data Recovery (DR) on the remote device. DES eliminates redundancy in the

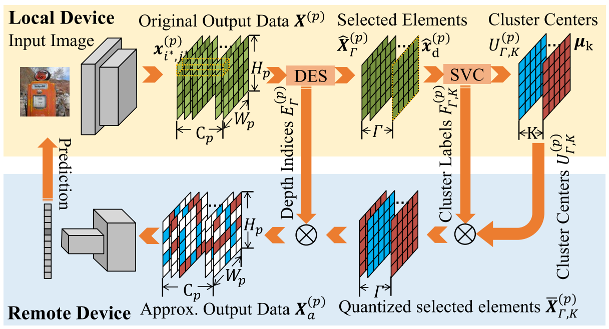
Local Device
Input Image
Original Output Data $X^{(p)}$
Selected Elements
Cluster Centers
DES
SVC
Depth Indices $E_\Gamma^{(p)}$
Cluster Labels $F_{\Gamma,K}^{(p)}$
Cluster Centers $U_{\Gamma,K}^{(p)}$
Prediction
Remote Device
Approx. Output Data $X_a^{(p)}$
Quantized selected elements $\tilde{X}_{\Gamma,K}^{(p)}$

Fig. 8. Workflow of FastFiltering compression.

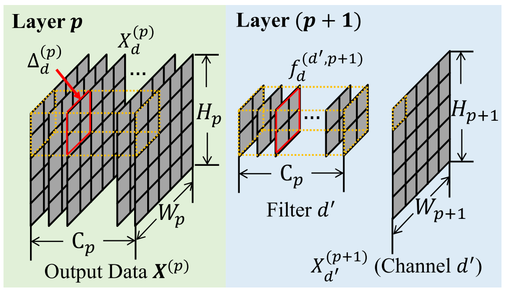

Fig. 9. Convolutional operation scheme in CNN models.

depth dimension, while SVC does the same across spatial planes. The DR module then restores the compressed data to its original size.

## A. DES Module

*Character of convolutional operation:* We propose DES based on the operating character of convolutional layer (over 90% of layers in a CNN models are of convolutional layer). As shown in Fig. 9, the output data of shape $H_p \times W_p \times C_p$ can be divided into $C_p$ 2D feature maps of shape $H_p \times W_p$. Specifically, we denote each 2D feature map as $X_d^{(p)}, d = 1, \ldots, C_p$. The element in $X_d^{(p)}$ is denoted as $x_{i,j,d}^{(p)}$, $i = 1, \ldots, H_p$, $j = 1, \ldots, W_p$. A convolutional filter $d'$ in the next layer $(p+1)$ consists of $C_p$ weight matrices as shown in Fig. 9. Each weight matrix of the filter $d'$ is denoted as $f_d^{(d',p+1)}$, $d = 1, \ldots, C_p$. An element in the $d'$-th feature map $X_{d'}^{(p+1)}$ in the output data from layer $(p+1)$ is computed by $\sum_{d=1}^{C_p} f_d^{(d',p+1)} \circledast \Delta_d^{(p)} + b_{d'}$, where the operation $\circledast$ is to multiply $f_d^{(d',p+1)}$ and $\Delta_d^{(p)}$ element-wisely and accumulate the multiplied results together, $b_{d'}$ is bias parameter of the filter $d'$, and $\Delta_d^{(p)}$ is the area on the $d$-th feature map in $\boldsymbol{X}^{(p)}$ that corresponds to the location of the element in $X_{d'}^{(p+1)}$. $\Delta_d^{(p)}$ is moved based on the stride parameter of layer $(p+1)$ to obtain the other elements in $X_{d'}^{(p+1)}$. The other filters in layer $(p+1)$ compute in the same way. In other words, *the elements at the same spatial position in the output of layer p are accumulated over all the channels to generate the output of layer (p+1).*

Such character of convolutional operation enlightens the proposal of DES module, in which we filter out elements with small absolute values across the depth dimension (all channels) to reduce the data size. As these elements have a minor contribution in the following accumulation across all channels in the next layer's operation, we can approximate the original intermediate output by setting them to zero. Specifically, we denote the elements at the same spatial position (e.g., $i^*, j^*$) across all the $C_p$ channels as $\boldsymbol{x}_{i^*,j^*}^{(p)} = \{x_{i^*,j^*,d}^{(p)}\}_{d=1,\ldots,C_p}$ shown in Fig. 8. Based on the discussion above, the elements in $\boldsymbol{x}_{i^*,j^*}^{(p)}$ can be redundant. In other words, not all elements of $\boldsymbol{x}_{i^*,j^*}^{(p)}$ are necessary for the correct prediction of a CNN model. The DES module is to select elements from each $\boldsymbol{x}_{i^*,j^*}^{(p)}$, $\{i^* = 1, \ldots, H_p\}$, $\{j^* = 1, \ldots, W_p\}$ based on their absolute values:

$$\hat{\boldsymbol{X}}_\Gamma^{(p)}, E_\Gamma^{(p)} = \{top(\boldsymbol{x}_{i^*,j^*}^{(p)}, \Gamma)\}_{i^* \in [1,H_p], j^* \in [1,W_p]} \quad (1)$$

where function $top(\cdot,\cdot)$ is to select $\Gamma$ maximum (absolute value) elements from $\boldsymbol{x}_{i^*,j^*}^{(p)}$, $\Gamma$ is a hyper-parameter that controls the trade-off between accuracy and transmission data size. As there are $C_p$ elements in $\boldsymbol{x}_{i^*,j^*}^{(p)}$, $\Gamma$ can be any integer from 1 to $C_p$. $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ is the set of selected elements from $\boldsymbol{X}^{(p)}$, i.e., the $\Gamma$ maximum elements in each spatial position $(i^*, j^*)$ across all the channels (i.e., $\boldsymbol{x}_{i^*,j^*}^{(p)}$). As the spatial dimension is $H_p \times W_p$, the shape of $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ is $H_p \times W_p \times \Gamma$. Each element of $E_\Gamma^{(p)}$ is the original depth (channel) index in $\boldsymbol{X}^{(p)}$ that each element in $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ locates. Thus, the shape of $E_\Gamma^{(p)}$ is $H_p \times W_p \times \Gamma$. One note is that, each element of $E_\Gamma^{(p)}$ is an integer ranging from 1 to $C_p$ and only needs $\lceil \log_2 C_p \rceil$ bits ($\lceil \cdot \rceil$ is the ceiling of $\log_2 C_p$) for storage and transmission. The latency overhead of DES module on the local device is $< 2$ ms based on our observation.

## B. SVC Module

In $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ (output from the DES module), there are $H_p \times W_p$ elements in a channel $d$ ($d = 1, \ldots, \Gamma$), denoted as $\hat{\boldsymbol{x}}_d^{(p)} = \{\hat{x}_{i,j,d}^{(p)}\}_{i \in [1,H_p], j \in [1,W_p]}$. Thus, $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ consists of $\Gamma$ vectors and each vector $\hat{\boldsymbol{x}}_d^{(p)}$ has $H_p \times W_p$ elements. In the SVC module, we use k-means clustering to separate these vectors into $K$ clusters $\boldsymbol{S} = \{S_1, S_2, \ldots, S_K\}$. Each vector $\hat{\boldsymbol{x}}_d^{(p)}$ is assigned to a cluster $S_k$ whose cluster center $\boldsymbol{\mu}_k$ is nearest to it, i.e., $\arg\min_{\boldsymbol{S}} \sum_{k=1}^{K} \sum_{\hat{\boldsymbol{x}}_d^{(p)} \in S_k} \|\hat{\boldsymbol{x}}_d^{(p)} - \boldsymbol{\mu}_k\|^2$, where $K$ is a hyper-parameter that controls the trade-off between accuracy and transmission data size. As there are $\Gamma$ vectors ($\hat{\boldsymbol{x}}_d^{(p)}$), $K$ can be any integer from 1 to $\Gamma$. With k-means clustering, the $\Gamma$ vectors in $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ are clustered to $K$ clusters, i.e., each vector is assigned to one of the $K$ clusters. We denote the set of cluster labels of the vectors as $F_{\Gamma,K}^{(p)}$. There are $\Gamma$ elements in $F_{\Gamma,K}^{(p)}$ and the $d$-th element in it represents the cluster label of the $d$-th vector in $\hat{\boldsymbol{X}}_\Gamma^{(p)}$, denoted as $k(d)$. As the cluster label ranges from 1 to $K$, each element of $F_{\Gamma,K}^{(p)}$ only needs $\lceil \log_2 K \rceil$ bits for storage and transmission. Furthermore, we denote the set of cluster centers as $U_{\Gamma,K}^{(p)} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$. The quantized $\hat{\boldsymbol{X}}_\Gamma^{(p)}$ is denoted as $\tilde{X}_{\Gamma,K}^{(p)}$, which has the same shape as $\hat{\boldsymbol{X}}_\Gamma^{(p)}$. Obviously, $\tilde{X}_{\Gamma,K}^{(p)}$

TABLE I
NUMBER OF BITS COMPARISON

| Data | Number of Bits |
|---|---|
| $\boldsymbol{X}^{(p)}$ | $32 \times H_p \times W_p \times C_p$ |
| $E_\Gamma^{(p)}$ | $\lceil \log_2 C_p \rceil \times H_p \times W_p \times \Gamma$ |
| $F_{\Gamma,K}^{(p)}$ | $\lceil \log_2 K \rceil \times \Gamma$ |
| $U_{\Gamma,K}^{(p)}$ | $32 \times H_p \times W_p \times K$ |

can be exactly constructed with $F_{\Gamma,K}^{(p)}$ and $U_{\Gamma,K}^{(p)}$. The latency overhead of SVC module on the local device ranges from 0.3 ms to 3.5 ms based on our observation.

## C. DR Module

As shown in Fig. 8, instead of sending the original output data $\boldsymbol{X}^{(p)}$, the local device sends the depth indices $E_\Gamma^{(p)}$, the cluster labels $F_{\Gamma,K}^{(p)}$, and the cluster centers $U_{\Gamma,K}^{(p)}$, to the remote device. The remote device then recovers an approximation of $\boldsymbol{X}^{(p)}$ based on them, denoted as $\boldsymbol{X}_a^{(p)}$ as follows: *Step-1:* Constructing $\tilde{X}_{\Gamma,K}^{(p)}$ based on $F_{\Gamma,K}^{(p)}$ and $U_{\Gamma,K}^{(p)}$. Note that $\tilde{X}_{\Gamma,K}^{(p)}$ is quantization of $\hat{\boldsymbol{X}}_\Gamma^{(p)}$. *Step-2:* Placing each element of $\tilde{X}_{\Gamma,K}^{(p)}$ in $\boldsymbol{X}_a^{(p)}$ at its original position in $\boldsymbol{X}^{(p)}$ according to the depth indices $E_\Gamma^{(p)}$. Specifically, the element of $E_\Gamma^{(p)}$ located at $(i^*, j^*, d^*)$ represents the depth index of $x_{i^*,j^*,d^*}^{(p)}$ (i.e., the element $x_{i^*,j^*,d^*}^{(p)}$ of $\tilde{X}_{\Gamma,K}^{(p)}$ located at $(i^*, j^*, d^*)$) in the original output data $\boldsymbol{X}^{(p)}$. The other elements in $\boldsymbol{X}_a^{(p)}$ are set to zeros, as shown in Fig. 8. The latency overhead of the DR module on the remote device is $< 4.5$ ms based on our observation.

## D. Transmission Data Size

Table I compares the number of bits of $\boldsymbol{X}^{(p)}$, $E_\Gamma^{(p)}$, $F_{\Gamma,K}^{(p)}$, and $U_{\Gamma,K}^{(p)}$. We assume that the elements in $\boldsymbol{X}^{(p)}$ and $U_{\Gamma,K}^{(p)}$ are of single-precision floating-point format (FP32) (default precision of CNN models' weights). As $H_p \cdot W_p \cdot C_p$ is larger than $10^4$ in most cases, $\lceil \log_2 K \rceil \cdot \Gamma \ll 32 \cdot H_p \cdot W_p \cdot C_p$. Thus, the ratio of data size after FF compression (i.e., compressed data size) to the original data size (i.e., original output data size) can be approximately calculated by $r_{FF} = \frac{\lceil \log_2 C_p \rceil}{32} \cdot \frac{\Gamma}{C_p} + \frac{K}{C_p}$. The $r_{FF}$ is in direct proportion to the hyper-parameters $\Gamma$ and $K$. Smaller $\Gamma$ and/or $K$ leads to lower $r_{FF}$ (higher compression rate), and vice versa. As the shape of $\boldsymbol{X}_a^{(p)}$ is the same as the shape of $\boldsymbol{X}^{(p)}$, $\boldsymbol{X}_a^{(p)}$ can be directly fed into the original layers of the CNN model on the server. Thus, FF implements transmission data compression without modifications to the CNN model. The hyper-parameters $\Gamma$ and $K$ determine the transmission data size and the information stored in the transmission data. Smaller $\Gamma$ (and/or $K$) leads to higher compression but preserves less information and vice versa.

## E. Optimization for Model Inference

In FF, the parameter $\Gamma$ plays a crucial role in balancing compression ratio and data accuracy. A smaller $\Gamma$ typically results in a higher compression ratio, reducing the size of data needed for transmission. However, this also means that more values in the original data are approximated to zeros, potentially leading to a loss in accuracy. Similarly, the parameter K exhibits comparable effects on performance, influencing both compression and accuracy. To address these challenges, we provide comprehensive guidance on how to optimally select $\Gamma$ and K in model inference.

In the offline stage, we profile the relationship between the model accuracy and the values of $\Gamma$ and K at each cutting layer $p$, denoted as $A(\Gamma, K; p)$. For each cutting layer $p$, we profile the processing latency on the local device as a function $L_1 = f_1(c_1; p)$, where $c_1$ is the contention on the local device. For neural networks, we use the GPU utilization rate as the contention [30]. Similarly, for each cutting layer $p$, we profile the processing latency on the remote device as a function $L_2 = f_2(c_2; p)$, where $c_2$ is the contention on the local device. Note that we ignore the computing latency of the modules in FF on the devices because we experimentally find that the computing latency of FF takes around 5% of the computing latency. Thus, we find that it does not deteriorate the performance when we ignore them in selecting $\Gamma$ and $K$.

As discussed in Section IV-D, the relationship between the transmission data size $S$ and the compression parameters $\Gamma$ and $K$ is also linear, denoted as $S(\Gamma, K; p) = a_1 \cdot \Gamma + b_1 \cdot K$, where $a_1 = \lceil \log_2 C_p \rceil \cdot H_p \cdot W_p$ and $b_1 = 32 \cdot H_p \cdot W_p$. Following [16], we approximate the the profiled relationships $A(\Gamma, K; p)$, $f_1(c_1; p)$, and $f_2(c_2; p)$ as linear functions. Specifically, we have $A(\Gamma, K; p) = a_2 \cdot \Gamma + b_2 \cdot K + c$, $f_1(c_1; p) = a_3 \cdot c_1 + b_3$, and $f_2(c_2; p) = a_4 \cdot c_2 + b_4$. We take the cutting layer $p$ as a parameter for brute-force search as other works on neurosurgery [5], [15]. Thus, for each $p$, we offline profile its corresponding set of $\{a_1, b_1; a_2, b_2, c; a_3, b_3; a_4, b_4\}$.

In the online stage, we design solutions of the optimal $\Gamma$ and $K$ for two cases. In the first case, the system requires the highest possible accuracy while adhering to the latency limits:

$$\max_{\Gamma,K,p} \; a_2 \cdot \Gamma + b_2 \cdot K + c, \tag{2}$$

$$s.t. \;\; \frac{a_1}{\theta} \cdot \Gamma + \frac{b_1}{\theta} \cdot K + C \leq T_{\max}, \tag{3}$$

$$1 \leq \Gamma \leq C_p, \tag{4}$$

$$K_{\min} \leq K \leq \Gamma. \tag{5}$$

where $C = a_3 \cdot c_1 + b_3 + a_4 \cdot c_2 + b_4$, $\theta$ is the observed network throughput, and $c_1$ and $c_2$ are the contentions of the local and the remote device, respectively. The $K_{\min}$ is a hyper-parameter and we can set it to any value above 1. For each cutting layer $p$, we find the optimal $\Gamma^{(p)} = \min\{\frac{\theta}{a_1}(T_{\max} - C) - \frac{b_1}{a_1} \cdot K_{\min}, C_p\}$ and $K^{(p)} = K_{\min}$ given $\frac{a_2}{a_1} > \frac{b_2}{b_1}$; $\Gamma^{(p)} = K^{(p)} = \min\{\frac{T_{\max} - C}{a_1 + b_1} \cdot \theta, C_p\}$ given $\frac{a_2}{a_1} < \frac{b_2}{b_1}$. The optimal $p^* = \arg\max_p\{a_2 \cdot \Gamma^{(p)} + b_2 \cdot K^{(p)} + c\}$ and the final solution is $\{\Gamma^{(p^*)}, K^{(p^*)}, p^*\}$.

In the second case, the system minimizes latency without compromising the specified accuracy threshold:

$$\min_{\Gamma,K,p} \; \frac{a_1}{\theta} \cdot \Gamma + \frac{b_1}{\theta} \cdot K + C \tag{6}$$

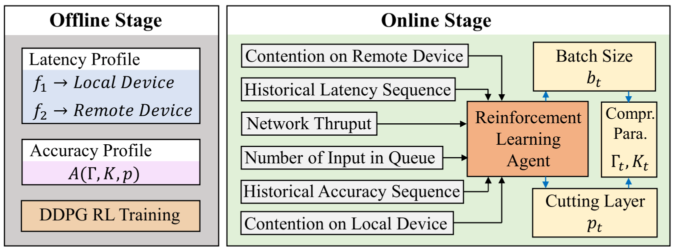

Fig. 10. Batch-pipeline-surgeon scheduler.

$$s.t.\ \ a_2 \cdot \Gamma + b_2 \cdot K + c \geq A_{\min}, \tag{7}$$

$$1 \leq \Gamma \leq C_p, \tag{8}$$

$$K_{\min} \leq K \leq \Gamma. \tag{9}$$

For each cutting layer $p$, we find the optimal $\Gamma^{(p)} = \min\{\frac{1}{a_2}(A_{\min} - c) - \frac{b_2}{a_2} \cdot K_{\min}, C_p\}$ and $K^{(p)} = K_{\min}$ given $\frac{a_2}{a_1} > \frac{b_2}{b_1}$; $\Gamma^{(p)} = K^{(p)} = \min\{\frac{A_{\min} - c}{a_2 + b_2}, C_p\}$ given $\frac{a_2}{a_1} < \frac{b_2}{b_1}$. The optimal $p^* = \arg\min_p\{\frac{a_1}{\theta} \cdot \Gamma^{(p)} + \frac{b_1}{\theta} \cdot K^{(p)} + C\}$ and the final solution is $\{\Gamma^{(p^*)}, K^{(p^*)}, p^*\}$.

## V. Batch-Pipeline-Surgeon Scheduler

In this section, we introduce a reinforcement learning-based scheduler, the BPS scheduler, aimed at enhancing continuous deep inference performance across local and remote devices. As depicted in Fig 10, the BPS scheduler rapidly determines the batch size, cutting layer, and compression parameters ($\Gamma$ and $K$ in FF compression) by analyzing current system conditions such as network throughput, device contentions, and queue length. The scheduler's RL agent is trained to produce scheduling decisions that effectively minimize the ratio of inference latency to accuracy.

### A. Problem Definition

As discussed in Section III-B, the rate of inference requests from users changes in a wide range. It is nontrivial to schedule the inferences of these requests to achieve high accuracy and low latency. We define the objective function as:

$$\min_{b_t, p_t} \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T} \frac{\bar{L}_t}{\bar{A}_t} \tag{10}$$

where $b_t$ is the batch size in each step, and $p_t$ is the cutting layer of the CNN model in each step; $\bar{L}_t$ is the average inference latency per request in each step, and $\bar{A}_t$ is the average accuracy per request in each step. Once the local device finishes its computation for inputs in the current step, the scheduler makes the decision for the next step.

### B. Calculation of Compression Parameters in FastFiltering

We can determine transmission compression parameters $\{\Gamma_t, K_t\}$ (in FF compression method) given current scheduling decision $\{b_t, p_t\}$ and previous scheduling decision $\{b_{t-1}, p_{t-1}\}$. Specifically, the computing latency on the local device for the inputs in current step can be predicted by a latency predictor [30], i.e., $L_{t,1} = f_1(b_t, c_1; p_t)$, where $c_1$ is the contention on the local device; Similarly, the computing latency on the remote device for the inputs in previous step can be predicted by $L_{t-1,2} = f_2(b_{t-1}, c_2; p_{t-1})$, where $c_2$ is the contention on the remote device; the transmission latency for the intermediate data in previous step can be estimated by $L_{t-1,d} = S(\Gamma_{t-1}, K_{t-1}, b_{t-1}; p_{t-1})/\theta$, where $S$ is the transmission data size for the inputs in previous step (can be easily calculated as shown in Section IV-D), $\theta$ is the network throughput. As shown in Fig. 11, if the intermediate data for the inputs in the current step arrive at the remote device at the exact time when the computation for the previous step finishes, then the scheduler fully utilizes the computing resources on the two devices (keeping them busy consistently). Thus, with $L_{t,1}$, $L_{t-1,2}$, and $L_{t,d}$, the optimal transmission latency for the inputs in current step is $L_{t,d} = L_{t-1,2} + L_{t-1,d} - L_{t,1}$. Given monitored network throughput $\theta$, the optimal transmission data size $S(\Gamma_t, K_t, b_t; p_t) = \theta \cdot L_{t,d}$. As described in Section IV-D, $S$ is linearly proportional to $\Gamma$ and $K$. Thus, we express $S(\Gamma_t, K_t, b_t; p_t)$ as $S(\Gamma_t, K_t, b_t; p_t) = a_1 \cdot \Gamma_t + b_1 \cdot K_t$, where $a_1 = \lceil \log_2 C_p \rceil \cdot H_p \cdot W_p \cdot b_t$ and $b_1 = 32 \cdot H_p \cdot W_p \cdot b_t$. Similarly, we approximate the relationship between accuracy and $\{\Gamma, K\}$ to be linear, i.e., $A(\Gamma_t, K_t; p_t) = a_2 \cdot \Gamma_t + b_2 \cdot K_t + c$, where $a_2, b_2, c$ can be measured offline [15], [30]. Consequently, the following problem is formulated:

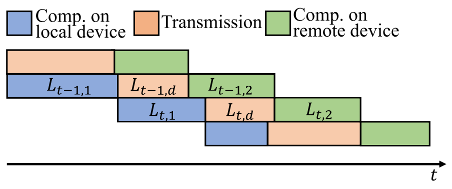

Fig. 11. Pipeline.

$$\max_{\Gamma_t, K_t} \ a_2 \cdot \Gamma_t + b_2 \cdot K_t + c, \tag{11}$$

$$s.t.,\ a_1 \cdot \Gamma_t + b_1 \cdot K_t = \theta \cdot L_{t,d}, \tag{12}$$

$$1 \leq \Gamma_t \leq C_{p_t}, \tag{13}$$

$$1 \leq K \leq \Gamma. \tag{14}$$

When $\frac{a_2 - a_1 b_2}{b_1} > 0$, $\Gamma_t = \min\{\lceil \frac{\theta \cdot L_{t,d} - b_1}{a_1} \rceil, C_{p_t}\}$ and $K_t = \max\{1, \lceil \frac{\theta \cdot L_{t,d} - a_1 \cdot \Gamma_t}{b_1} \rceil\}$; When $\frac{a_2 - a_1 b_2}{b_1} < 0$, $\Gamma_t = \max\{\lceil \frac{\theta \cdot L_{t,d} - b_1}{a_1} \rceil, 1\}$ and $K_t = \min\{\Gamma_t, \lceil \frac{\theta \cdot L_{t,d} - a_1 \cdot \Gamma_t}{b_1} \rceil\}$. The solution of $\Gamma_t$ and $K_t$ only contains simple algebra operations and can be finished in neglectable time on the local device. In this way, we obtain all the parameters $\{b_t, p_t, \Gamma_t, K_t\}$ for batching, pipelining, and neurosurgeon in the current scheduling step $t$.

### C. BPS Scheduler

As shown in Fig. 10, we utilize a reinforcement learning algorithm to train a BPS scheduler. Specifically, the RL agent makes decisions on two parameters $b_t, p_t$ at each step $t$, i.e., its action $a_t = (b_t, p_t)$. The RL agent observe the system state

$s_t = (\theta_t, Q_t, c_{t,1}, c_{t,2}, \mathbf{A}_t, \mathbf{L}_t)$, where $\theta_t$ is current network throughput, $Q_t$ is the number of inputs in the queue, $c_{t,1}$ is the contention on the local device, $c_{t,2}$ is the contention on the remote device, $\mathbf{A}_t$ is the sequence of historical average accuracies in previous steps, and $\mathbf{L}_t$ is the sequence of historical average latencies in previous steps. To train the RL agent, we utilize DDPG method [31], as shown in Algorithm 1. The reward function is defined as: $r = -\bar{L}_t/\bar{A}_t$. We utilize the Ornstein-Uhlenbeck process as the random action exploration $\mathcal{N}$ (line 5 in Algorithm 1). We set the number of episodes as 300 (line 4 in Algorithm 1). The hyper-parameter $N$ as 50 (line 12 in Algorithm 1) and $\tau$ as 0.005 (line 20 in Algorithm 1). The original two outputs from the actor network in the RL agent range between 0 and 1. We map one of them to batch size $b_t$ by multiplying it with $B$ (maximum batch size[1]), and the other to cutting layer $p_t$ by multiplying it with $|\mathcal{M}|$ (the number of layers in the CNN model $\mathcal{M}$), respectively (line 9 in Algorithm 1). In each step, the actor network $\mu$ generates the action based on the observed state and random action exploration (line 8 to 10 in Algorithm 1). We apply the (mapped) action to the system and observe the new state and the reward (line 11 in Algorithm 1). The tuple of states, reward, and action is stored into the replay buffer $R$ (line 12 in Algorithm 1). At each step, we randomly select $N$ tuples from the buffer (line 13 in Algorithm 1) to train the critic network (line 15 and 16 in Algorithm 1) and the actor network (line 17 and 18 in Algorithm 1). The critic network is trained by minimizing the difference between the predicted Q-value $Q(s_i, \bar{a}_i|\theta_Q)$ and the target Q-value calculated using the target networks $Q'$ and $\mu'$ (line 14 in Algorithm 1) At the end of each step, we softly update the target networks using the $\tau$ parameter (line 20 in Algorithm 1).

**Algorithm 1:** Training BPS Scheduler w/ DDPG Method.

```
1: Initialize critic network $Q(s,a|\theta_Q)$ and actor network
   $\mu(s|\theta_\mu)$ with weights $\theta_Q$ and $\theta_\mu$;
2: Initialize target networks $Q'$ and $\mu'$ with weights
   $\theta_{Q'} \leftarrow \theta_Q$ and $\theta_{\mu'} \leftarrow \theta_\mu$;
3: Initialize replay buffer $R$.
4: For $episode = 1, 2, \ldots$
5:   Initialize random action exploration $\mathcal{N}$;
6:   Receive initial observation state $s_1$;
7:   For $t = 1, 2, \ldots$
8:     $\bar{a}_t = \mu(s_t|\theta_\mu) + \mathcal{N}_t$;
9:     Map $\bar{a}_t[0]$ to batch size $b_t$ and $\bar{a}_t[1]$ to layer $p_t$;
10:    Obtain $a_t = (b_t, p_t)$;
11:    Execute $a_t$ and observe reward $r_t$ and $s_{t+1}$;
12:    Store transition $(s_t, \bar{a}_t, r_t, s_{t+1})$ in $R$;
13:    Sample $N$ transitions of $(s_i, \bar{a}_i, r_i, s_{i+1})$ from $R$;
14:    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta_{\mu'})|\theta_{Q'})$;
15:    Update the critic $Q(s, \bar{a}|\theta_Q)$ by minimizing:
16:    $L = \frac{1}{N}\sum_i (y_i - Q(s_i, \bar{a}_i|\theta_Q))^2$;
17:    Update the actor $\mu(s|\theta_\mu)$ by policy gradient:
18:    $\nabla_{\theta_\mu} J \approx$
       $\frac{1}{N}\sum_i \nabla_a Q(s, \bar{a}|\theta_Q)|_{s_i,\mu(s_i)} \nabla_{\theta_\mu}\mu(s|\theta_\mu)|_{s_i}$;
19:    Update the target networks:
20:    $\theta_{Q'} = \tau\theta_Q + (1-\tau)\theta_{Q'}, \theta_{\mu'} = \tau\theta_\mu + (1-\tau)\theta_{\mu'}$.
```

## VI. Evaluation

In this section, we evaluate the performance of FF compression and BPS scheduler.

*Hardware:* For the local device, we have two types of devices: Jetson Nano and Raspberry Pi4. For the remote device, we have two types of devices: Jetson TX2 and Jetson Xavier. The network connection between the local device and the remote device is WiFi 2.4 GHz, and the users connect to the local device through cables.

*CNN models:* We include seven CNN models: VGG-16 [3], ResNet-50 [32], Inception-V3 [33], MobileNet-V3 [34], tiny-YOLOv3 [35], SSD [36], FasterRCNN [37].

*Metrics: For the FF compression*, we have three metrics: accuracy loss, data size after compression, and decompression latency. *For the BPS scheduler*, we use the ratio of inference accuracy over processing latency as the metric. We use the average inference accuracy and processing latency per request. Note that a higher ratio indicates higher accuracy with lower processing latency.

*Baselines: For the FF compression*, we have SOTA lossy [20], [21] and DeepCOD+ as baselines. For SOTA lossy, we choose the one that achieves a higher compression rate between the JPEG-based compression (accelerated by nvJPEG [38]) and the Huffman Coding [20], [21]. For *DeepCOD* [15], we compress its decoder with SOTA model compression method [29] (which generates 0.21% to 10.75% extra accuracy loss and 2.4% to 66.3% latency reduction), denoted as DeepCOD+. *For the BPS scheduler*, we have the following baselines: (1) Local-RL: We process all the inputs on the local device. We train an RL agent to make the decision on the batch size. (2) Offload-RL: We offload all the inputs to the remote device in the network. We train an RL agent to make the decision on the batch size. (3) Data Parallel-RL: We train an RL agent to make the decision on the batch sizes on the local and remote devices. (4) Greedy: We make the decision on the batch size and the cutting layer greedily. The greedy method determines the batch size and the cutting layer based on the current network throughput ($\theta_t$), the queue length of inputs ($Q_t$), contentions of devices ($c_{t,1}$ and $c_{t,2}$), directly. Specifically, we set $b_t = \min\{Q_t, B\}$ and $p_t = \arg\min_p(\bar{L}_t/\bar{A}_t)$. Following [5], we brute-forcely compute $\bar{L}_t$ and $\bar{A}_t$ by varying $p \in [1, |\mathcal{M}|]$ and find the optimal $p$ value that shows the lowest $\bar{L}_t/\bar{A}_t$. The greedy method optimizes for immediate rewards, making choices that seem best at the current step without considering the long-term consequences. In contrast, reinforcement learning is designed to optimize long-term rewards. It uses the concepts of policy (a strategy to choose actions) and value functions (which estimate the long-term rewards of states), offering a more nuanced approach to decision-making compared to the single-step optimization in greedy methods. (5) BPS-DeepCOD+: We substitute FF with DeepCOD+ in the BPS scheduler. (6) BPS-lossy: We substitute FF with SOTA lossy in the BPS scheduler.

[1]Note that, though $B$ can be arbitrarily set, a large value may lead to a potentially high risk of memory overflow in execution.

TABLE II
COMPARISON OF COMPRESSION METHODS ON VGG-16 (NANO-TX2, LATENCY UNIT: MS)

| Compr Method | Conv2.1 | | | | | Conv3.1 | | | | | Conv4.1 | | | | | Conv5.1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ |
| FF | 135KB (4.3%) | **84.4% (-6.0%)** | 1.8 - | 21.0 - | 2.3 - | 91.4KB (5.8%) | **89.3% (-1.1%)** | 2.0 - | 16.2 - | 2.7 - | 25.2KB (3.2%) | **89.4% (-1.0%)** | 2.4 - | 6.8 - | 3.1 - | 6.9KB (1.8%) | **89.7% (-0.7%)** | 2.9 - | 5.2 - | 3.4 - |
| Deep COD+ | **2.3KB (0.07%)** | 84.3% (-6.1%) | 2.1 - | 4.2 - | 25.4 - | **1.7KB (0.11%)** | 85.1% (-5.3%) | 2.1 - | 3.9 - | 30.5 - | **988B (0.13%)** | 84.9% (-5.5%) | 2.7 - | 3.7 - | 37.8 - | **469B (0.12%)** | 85.6% (-4.1%) | 3.4 - | 3.5 - | 40.6 - |
| SOTA Lossy | 379KB (12.1%) | 76.6% (-13.8%) | 1.7 - | 37.6 - | 0.8 - | 185KB (11.8%) | 82.0% (-8.4%) | 1.9 - | 25.9 - | 1.0 - | 87.0KB (11.1%) | 83.8% (-6.6%) | 2.2 - | 14.2 - | 1.5 - | 78.0KB (19.9%) | 88.2 (-2.2%) | 2.6 - | 13.2 - | 2.4 - |
| Original | 3136KB | 90.4% | 0 | 325.3 | 0 | 1568KB | 90.4% | 0 | 178.6 | 0 | 784KB | 90.4% | 0 | 83.4 | 0 | 392KB | 90.4% | 0 | 49.3 | 0 |

TABLE III
COMPARISON OF COMPRESSION METHODS ON RESNET-50 (NANO-TX2, LATENCY UNIT: MS)

| Compr Method | Block 1 | | | | | Block 2 | | | | | Block 3 | | | | | Block 4 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ | Size | Acc | $t_{enc}$ | $t_{tr}$ | $t_{dec}$ |
| FF | 123.5KB (15.7%) | 83.7% (-9.2%) | 1.5 - | 19.4 - | 2.1 - | 127.0KB (8.1%) | **88.8% (-4.1%)** | 1.8 - | 19.4 - | 2.3 - | 46.3KB {(5.9%) | **89.9% (-2.9%)** | 2.1 - | 8.5 - | 2.6 - | 6.7KB (1.7%) | **90.6% (-2.2%)** | 2.6 - | 5.2 - | 3.0 - |
| Deep COD+ | **980B (0.12%)** | **84.7% (-8.1%)** | 2.0 - | 3.7 - | 28.7 - | **245B (0.02%)** | 87.0% (-5.8%) | 2.3 - | 3.2 - | 36.2 - | **184B (0.02%)** | 86.3% (-6.5%) | 2.6 - | 2.8 - | 42.7 - | **123B (0.03%)** | 86.6% (-6.2%) | 3.3 - | 2.5 - | 47.5 - |
| SOTA Lossy | 94.7KB (12.1%) | 77.1% (-15.8%) | 1.2 - | 16.9 - | 0.7 - | 178KB (11.4%) | 79.0% (-13.9%) | 1.5 - | 25.1 - | 0.9 - | 87.4KB (11.1%) | 86.6% (-6.3%) | 1.9 - | 14.2 - | 1.2 - | 80.8KB (20.6%) | 89.1% (-4.2%) | 2.4 - | 13.4 - | 1.5 - |
| Original | 784KB | 92.8% | 0 | 83.4 | 0 | 1568KB | 92.8% | 0 | 178.6 | 0 | 784KB | 92.8% | 0 | 83.4 | 0 | 392KB | 92.8% | 0 | 49.3 | 0 |

*Latency Predictor:* For the latency predictors ($f_1$ and $f_2$) in BPS scheduler, we use quadratic regression models to approximate the latency of CNN inference. In the offline stage, we train the predictors by varying the batch size, cutting layer, and device contention. The prediction accuracy can reach $> 96\%$ on the devices in our evaluation, and the online prediction overhead is neglectable.

*Design and train of RL:* During training, the learning rates of the actor and critic network as $10^{-4}$ and $10^{-3}$, respectively. The critic network consists of four fully-connected layers with dimensions of $\{200, 100, 50, 50\}$ and the actor network consists of three fully-connected layers with dimensions of $\{40, 20\}$. The length of historical accuracies and latencies is five, respectively. Note that we only need the actor network in the online stage, and the actor network generates neglectable overhead on the local device. We train the RL agent offline for 12 hours on a workstation with one RTX2080 GPU. For object detection service, five cameras stream live video (keyframes) to the local device for object detection. The videos are from webcams from [39]. For image recognition, up to eight users randomly send images (from ImageNet validation dataset [40]) to the local devices, the request rate follows Poisson inter-arrival time distribution [18].

## A. Performance of FastFiltering

In Table II, we offload the output from {'pool1', 'pool2', pool3', 'pool4'} of VGG-16 to the server. Correspondingly, the first layer on the server is {'conv2.1', 'conv3.1', 'conv4.1', 'conv5.1'} of VGG-16. In Table III, we partition ResNet-50 before each block like [15]. For FF, at each cutting layer, we tune the compression parameters ($\Gamma$ and $K$) to find the optimal ones that achieve the highest ratio of accuracy over latency. For SOTA lossy, we also find the optimal compression point. For DeepCOD+, we compress its decoder to the optimal level that also achieves the highest ratio of accuracy over latency. The compression performances (data size after compression, accuracy, latency overhead) with the optimal parameters in each cutting layer are shown in Tables II and III. The latency overhead is consisted of the compression latency ($t_{enc}$), the transmission latency ($t_{tr}$), and the decompressing latency ($t_{dec}$).

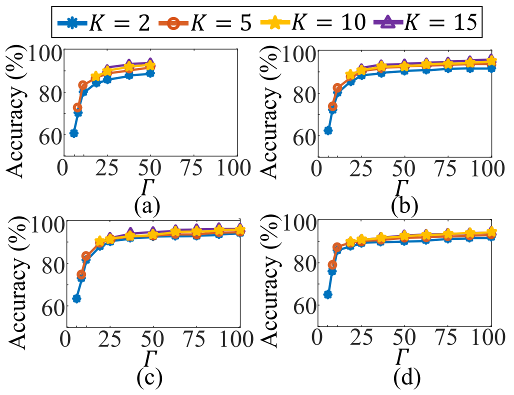

Fig. 12. Accuracy (Top-5) of VGG-16 w/ Different $\Pi$, $\Gamma$, and $K$: (a) $\Pi = 5$; (b) $\Pi = 10$; (c) $\Pi = 19$; (d) $\Pi = 28$.

For the compression, the computational complexity of k-means clustering (the DES module) is proportional to $\Gamma$ (number of features) and $K$ (number of cluster centers). However, as shown in Fig. 12, raising $K$ above 5 does not contribute significantly to the accuracy improvement. Thus, we can keep $K$ below 5 with trivial accuracy reduction. Similarly, we can keep $\Gamma$ below 40 with trivial accuracy reduction. Furthermore, as the edge devices are equipped with embedded GPUs and/or multi-core CPUs, we further reduce the computing latency by paralleling the computation of the clustering. Based on our observation, the overhead of k-means clustering accounts for around 65% in FF compression, which is less than 2 ms.

As shown in Tables II and III, FF compression achieves the highest accuracy (lowest accuracy loss) and the lowest latency overhead simultaneously in most cases. The accuracy loss with FF compression reaches 17.2% of that with DeepCOD+ (VGG-16 at Conv5.1), and reaches 13.1% of that with SOTA lossy (VGG-16 at Conv3.1). The latency overhead with FF compression reaches 16.4% of that with DeepCOD+ (ResNet $= 50$ at Block 4), and reaches 60.7% of that with SOTA lossy (VGG-16 at Conv2.1). We explain the underlying reason why our FF compression methods outperform DeepCOD+ and SOTA lossy. *Compared* to SOTA lossy, FF method exploits the operating character of convolution layers (Section IV-A) and efficient approximation (Section IV-A and IV-B). By preserving key information in the intermediate output data effectively, up to 18.9% higher compression ratio is obtained by our FF method, which leads to 61.2% transmission latency reduction compared to SOTA lossy. In addition, the decompressing latency of our FF method is only 1.0 ms to 1.7 ms higher than that of SOTA lossy. Consequently, the overall overhead (accumulated latency of transmission and decompressing latency) of our FF method is up to 83.6% lower than that of SOTA lossy. By designing the compression based on the character of convolutional operation, our FF method also outperforms SOTA lossy regarding the accuracy, i.e., its accuracy loss is as low as 13.1% of that with SOTA lossy.

*Compared* to DeepCOD+, our DR module recovers the compressed data in a much simpler way, which ensures efficiency on relatively low-performance remote devices. DeepCOD, designed specifically for device-server neurosurgeon workflows, implements an asymmetric encoding-decoding framework that predominantly burdens the server side, as outlined in [15]. In their design, a lightweight encoder compresses data on the local device, while a more complex decoder is responsible for data reconstruction on the remote server. The decoder in DeepCOD employs a content generation technique to enhance the data dimensionality, which significantly increases the number of computational operations. Consequently, in situations where the remote server or devices have limited capabilities, DeepCOD's decoder experiences extended decompression latency significantly. On the other hand, our approach to decompression involves reconstructing data using cluster labels and depth indices. This method maintains a considerably lower count of operations compared to DeepCOD, enhancing efficiency, especially in resource-constrained environments. Though DeepCOD+ achieves an extreme small transmission data size (e.g., $< 1$KB), its reduction in transmission latency is minor compared to its high decoding latency on the remote device. Even a compressed-version decoder with high accuracy loss (3.4% to 5.2%) generates $> 10\times$ longer decompressing latency compared to our FF method. It is important to address that, as WiFi 2.4 GHz (450 Mbps to 600 Mbps) has a much smaller bandwidth than popular wireless networks (e.g., WiFi 5 GHz, up to 1300 Mbps) today, our setup actually favors DeepCOD+ whose compression ratio is high. Evaluating the performance in a better network like WiFi 5 GHz is expected to increase the superiority of our FF method over DeepCOD+; meanwhile, the superiority of our FF over SOTA lossy is irrelevant to network conditions.

## B. Performance of BPS Scheduler

We evaluate the performance of our BPS scheduler (Section V) in various cases. In Fig. 13, we evaluate the performance of BPS in image recognition (VGG-16) and object detection (tiny-YOLOV3) on four pairs of local and remote devices. Compared to local-RL, BPS-FF borrows the computing resources from the remote device and outperforms local-RL by $7.3\times$ on average. Compared to offload-RL, BPS-FF utilizes both local and remote devices for deep inference and outperforms offload-RL by $2.7\times$ on average. Compared to DataParallel-RL, BPS-FF utilizes neurosurgeon and FF techniques to adjust the portion of computing load across the two devices and outperforms DataParallel-RL by $1.8\times$ on average. Compared to Greedy-FF, BPS-FF adopts reinforcement learning technique to learn the underlying pattern of the relationship between system performance and scheduling decisions and outperforms greedy-FF by $1.4\times$ on average. Compared to BPS-DeepCOD+ and BPS-Lossy, BPS-FF takes benefits from the FF compression (i.e., high-granularity curve of accuracy-latency trade-off and Pareto optimality) and outperforms BPS-DeepCOD+ by $2.6\times$, BPS-Lossy by $1.7\times$. In Fig. 14, we further observe the performance of BPS-FF on other CNN models and compare it with the baseline schedulers. Similar to the discussion above, BPS-FF outperforms the baselines by wisely utilizing the computing resources on local and remote devices. Overall, it outperforms the baselines by 8.2 to $1.4\times$.

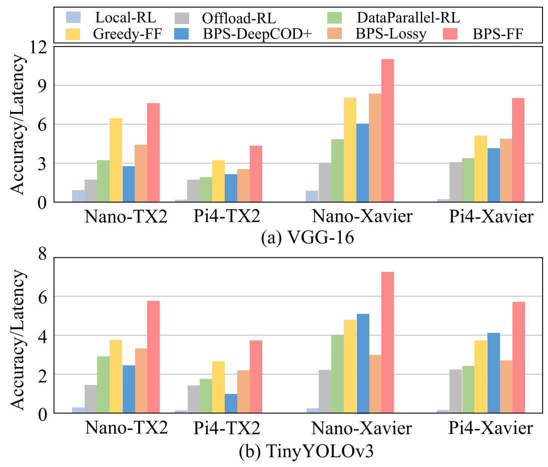

Fig. 13. BPS performance in: (a) image recognition, (b) object detection.

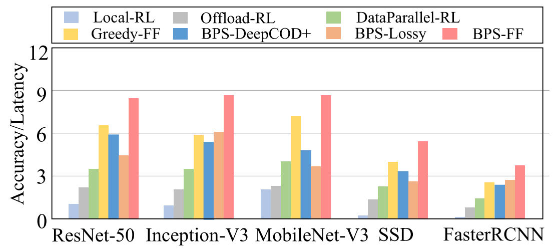

Fig. 14. BPS performance with different models (Nano-TX2).

We closely examine a 10-second transient performance, as shown in Fig. 15. Within this timeframe, we note a fluctuation in the number of requests: initially averaging about 80 every 300 ms, dropping to approximately 20 at 23.6 seconds, and then climbing back to around 70 at 26.3 seconds. The conventional greedy method determines batch size ($b_t$) and cutting (partition) layer ($p_t$) based solely on the current network status, request queue length, and device contention, without accounting for the long-term dependencies between temporal states. In contrast, the BPS scheduler employs a learned policy that optimizes

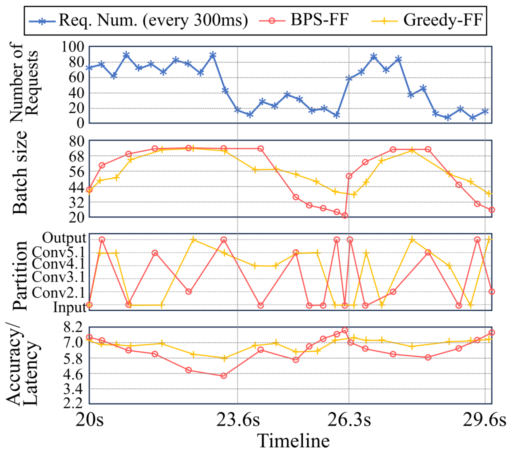

Fig. 15. Transient observation (Nano-TX2).

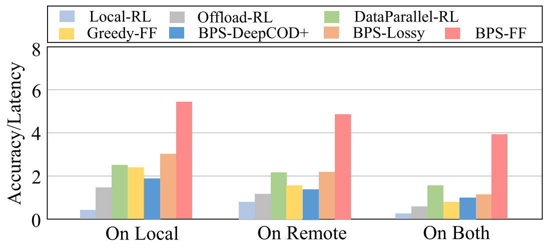

Fig. 16. Performance with real background app on device(s) (Nano-TX2).

for long-term performance by considering these dependencies. During the transient phase shown in Fig. 15, we pay particular attention to the system's response to changes in the number of requests, which directly relates to the request queue length ($A_t$). Under high request loads, such as from 20 s to 23.6 s, the greedy method reacts immediately by increasing the batch size, thus compromising accuracy to reduce processing latency. This approach results in a noticeable drop in the performance (Accuracy/Latency) during periods of intense requests. Its performance recovers during times of lower request density, e.g., from 23.6 s to 26.3 s, as the system only needs to process a smaller influx of requests, allowing for high-quality processing with greater accuracy and lower latency. On the other hand, the BPS scheduler adopts a more consistent approach to handling requests. During high-demand periods (e.g.,, from 20 s to 23.6 s), unlike the greedy method, the BPS scheduler does not significantly increase batch size. Instead, it strategically allows some requests to be processed later, during less busy periods (e.g., from 23.6 s to 26.3 s). As a result, as shown in Fig. 15, the batch size under the BPS scheduler remains smaller than the greedy method during peak request times and larger during quieter intervals. This strategy, informed by an awareness of long-term dependencies, also aids the BPS scheduler in choosing parameters that maintain high accuracy, particularly under heavy request loads. Consequently, as shown in Fig. 15, the ratio of accuracy to latency performance maintained by the BPS scheduler consistently surpasses that of the greedy method during these high-demand periods.

### C. Performance With Background On-Device Applications

We further evaluate the performance of the BPS scheduler under a real background application running on the local/remote device(s). Following the test approach in [30], we run a GPU-intensive linear algebra routine (Gaussian Elimination from the Rodinia Benchmark [41]). We run the background application while keeping the inference service for 10 minute. As shown in Fig. 16, in all three cases (background app on the device, on the remote device, and both devices), BPS-FF outperforms the baseline schedulers by 4.3× on average. Besides the reasons described in Section VI-B, the superiority of BPS-FF over the baselines is also achieved by the high-granularity online-tunable FF compression, which allows the scheduler to adaptively adjust its scheduling decisions according to the change of contention on the device(s).

## VII. Related Work

In the realm of edge computing, Neurosurgeon [5] proposes a method to reduce computational load on edge/mobile devices by offloading part of a CNN model to a high-performance edge/cloud server. This approach involves partitioning the model at an intermediate layer, processing early layers locally and transmitting the intermediate output for the remaining layers to a server. Subsequent studies have expanded on Neurosurgeon, exploring intermediate data compression [15], video analytics [16], and integration with model compression [14]. Meanwhile, compression of the intermediate output is pivotal in Neurosurgeon-based offloading. Techniques like JPEG-based compression and Huffman coding have been employed [20], [21], but their effectiveness on CNN intermediate outputs is limited, often resulting in large data sizes and significant accuracy loss [15]. The method proposed in [15] involves adding extra encoding and decoding layers at partition points, achieving higher compression rates with lower accuracy loss but necessitating extensive additional training for each potential partition point, which can be time-consuming.

In server cluster parallel processing, Hone [42] introduces a tuple scheduler using an online Largest-Backlog-First strategy to minimize the maximum queue backlog, thereby reducing latency in stream processing. Parrot [43] offers an online coflow-aware framework, optimizing the scheduling of dependent coflows in distributed machine learning jobs to decrease overall job completion times in shared clusters. Our approach with the BPS scheduler is distinct, aiming to optimize deep inference in tandem between local and remote edge devices. BPS combines efficient intermediate data compression with adaptive techniques like batching, pipelining, and surgeon to improve performance. Unlike Hone, which manages stragglers in large clusters, and Parrot, which is dedicated to machine learning cluster job scheduling, BPS is tailored for the specific demands of collaborative edge computing.

## VIII. Discussion

*FF Advantages:* Compared to basic compression schemes that simply discard small values, FF offers several advanced features. Primarily, FF employs its the DES module to selectively discard small values, with a specific focus on the depth dimension.

This targeted selection aligns with the convolutional operation's intrinsic mechanism, where elements at the same spatial location in a layer's output are cumulatively processed across all channels for the next layer's output. This depth-oriented approach contrasts with general compression schemes, which may not consider discarding small values in the depth dimension, as they typically do not account for the unique operational dynamics of convolutional processes. For instance, traditional image compression methods primarily focus on spatial dimensions (height $\times$ width). Additionally, FF's SVC module further refines the data reduction process, going beyond the capabilities of the DES module. The SVC module employs K-means clustering over the depth dimension, distinctively approximating the layer output data already refined by the DES module. This approach allows the SVC module to effectively handle data devoid of smaller values, a level of data approximation not typically achieved by simple compression schemes that merely discard small values. By integrating these modules, FF demonstrates a nuanced understanding of data characteristics in CNN models, enabling more effective and model-specific compression than general methods.

*Model Compression Versus FF:* In the realm of neural network model compression, quantization is a technique that decreases the precision of a model's parameters. This process involves converting each parameter from a high-precision format, such as 32-bit, to a lower precision format, like 8-bit. On the other hand, FF focuses on reducing the size of the output data of a neural network layer. FF achieves this by employing methods such as filtering and clustering feature maps across the channels of the network. While quantization enhances a model's efficiency by lowering the precision of its parameters, FF directly diminishes the volume of output data from a neural network layer by decreasing the number of its channels. It is important to note that the application of quantization and pruning techniques would not diminish the advantages offered by FF. These techniques primarily aim to reduce computational latency by decreasing the number of operations required by the models. In contrast, FF is designed to reduce the data size of the layer outputs. Therefore, FF functions as a data compression method for the intermediate data in convolutional neural networks, complementing quantization and pruning. The latter are model compression methods that optimize the structure of convolutional neural networks. By targeting different aspects of efficiency – FF focusing on data size reduction and quantization/pruning on operational efficiency – these approaches can be effectively combined to enhance overall network performance.

*Scalability of FF and BPS:* Both the FF compression technique and the BPS scheduler can be modified to scenarios involving more than two devices. For FF compression, the method can be implemented across each data transmission occurring between any two devices in the network. In the case of the BPS scheduler, adaptations can be made to the state and action definitions within its reinforcement learning framework. Specifically, this involves augmenting the state to include the contention levels of all devices and the network throughput for each device pair. As for the action space, it expands to encompass the decisions regarding the cutting layer for each device, represented as $\{p_t^{(0)}, p_t^{(1)}, \ldots, p_t^{(N)}\}$, where $N$ is the total number of devices in the system. In practical applications, we ensure $p_t^{(i)} = \max\{p_t^{(i)}, p_t^{(i-1)}\}$ to maintain that the cutting layer index for the $i$-th device is always greater than or equal to that of the preceding device. If a situation arises where $p_t^{(i)} \leq p_t^{(i-1)}$, it implies that there will be no model execution on the $i$-th device.

*System Decentralization:* The system can operate in a decentralized manner, allowing the remote device to manage its own inference tasks independently. This flexibility is facilitated by the BPS scheduler, which considers the contention levels of the devices, such as their GPU utilization rates. As a result, the remote device, influenced by its own contention state, can effectively execute its inference tasks while concurrently running a BPS scheduler to optimize its computational processes. When both devices are performing inference tasks simultaneously, it becomes essential for their respective BPS schedulers to coordinate. One possible approach is to establish a priority system, where one scheduler's decisions take precedence, and the other scheduler adapts its actions based on the leading scheduler's decisions. Furthermore, integrating multi-agent reinforcement learning concepts could enhance this coordination. By treating each BPS scheduler as an individual agent, they can collaboratively work towards optimizing a collective reward. This reward could be framed in terms of maximizing overall accuracy and/or minimizing the average latency across the inference tasks on both devices. Such a collaborative approach ensures that while each device operates autonomously, they collectively contribute to the system's overarching performance goals.

## IX. Conclusion

Responding to the growing need for efficient deep inference at the edge, this paper addresses the scheduling challenges posed by the availability of a remote device for shared computational tasks. We introduced BPS, an adaptive online scheduler designed for collaborative edge intelligence. BPS incorporates data parallel, neurosurgeon, and reinforcement learning techniques to enhance inference performance, achieving an up to $8.2\times$ improvement over traditional schedulers. Additionally, we developed FF, a specialized compressor for intermediate output data in neurosurgeon applications. FF capitalizes on the unique aspects of convolutional layers and employs effective approximation algorithms, resulting in up to 86.9% less accuracy loss and 83.6% less latency overhead compared to existing compression methods. Together, the BPS scheduler and FF compressor offer a robust solution for optimizing continuous deep inference in collaborative edge environments.

## References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[4] K. Bhardwaj, C.-Y. Lin, A. Sartor, and R. Marculescu, “Memory-and communication-aware model compression for distributed deep learning inference on IoT,” *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–22, 2019.

[5] Y. Kang et al., “Neurosurgeon: Collaborative intelligence between the cloud and mobile edge,” *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.

[6] R. D. Evans, L. Liu, and T. M. Aamodt, “JPEG-ACT: Accelerating deep learning via transform-based lossy compression,” in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Architecture*, 2020, pp. 860–873.

[7] D. Gudovskiy, A. Hodgkinson, and L. Rigazio, “DNN feature map compression using learned representation over GF (2),” in *Proc. Eur. Conf. Comput. Vis. Workshops*, 2018, pp. 502–516.

[8] J.-H. Luo and J. Wu, “An entropy-based pruning method for CNN compression,” 2017, *arXiv: 1706.05791*.

[9] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, “BottleFit: Learning compressed representations in deep neural networks for effective and efficient split computing,” 2022, *arXiv:2201.02693*.

[10] Y. Shi, M. Wang, S. Chen, J. Wei, and Z. Wang, “Transform-based feature map compression for CNN inference,” in *Proc. IEEE Int. Symp. Circuits Syst.*, 2021, pp. 1–5.

[11] S. I. Young, Z. Wang, D. Taubman, and B. Girod, “Transform quantization for CNN compression,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 9, pp. 5700–5714, Sep. 2022.

[12] X. Hou, Y. Guan, T. Han, and N. Zhang, “DistrEdge: Speeding up convolutional neural network inference on distributed edge devices,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 1097–1107.

[13] E. Baccour, A. Erbad, A. Mohamed, M. Hamdi, and M. Guizani, “RL-PDNN: Reinforcement learning for privacy-aware distributed neural networks in IoT systems,” *IEEE Access*, vol. 9, pp. 54872–54887, 2021.

[14] M. Krouka, A. Elgabli, C. B. Issaid, and M. Bennis, “Energy-efficient model compression and splitting for collaborative inference over time-varying channels,” in *Proc. IEEE 32nd Annu. Int. Symp. Pers., Indoor Mobile Radio Commun.*, 2021, pp. 1173–1178.

[15] S. Yao et al., “Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency,” in *Proc. 18th Conf. Embedded Netw. Sensor Syst.*, 2020, pp. 476–488.

[16] L. Zhang, L. Chen, and J. Xu, “Autodidactic neurosurgeon: Collaborative deep inference for mobile edge intelligence via online learning,” in *Proc. Web Conf.*, 2021, pp. 3111–3123.

[17] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A Low-Latency online prediction serving system,” in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 613–627.

[18] A. Gujarati et al., “Serving DNNs like clockwork: Performance predictability from the bottom up,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 443–462.

[19] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, “INFaaS: Automated model-less inference serving,” in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 397–411.

[20] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, “Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms,” in *Proc. IEEE 15th Int. Conf. Adv. Video Signal Based Surveill.*, 2018, pp. 1–6.

[21] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, “JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution,” in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, 2018, pp. 671–678.

[22] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, “Chameleon: Scalable adaptation of video analytics,” in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 253–266.

[23] L. Liu, H. Li, and M. Gruteser, “Edge assisted real-time object detection for mobile augmented reality,” in *Proc. 25th Annu. Int. Conf. Mobile Comput. Netw.*, 2019, pp. 1–16.

[24] X. Wang, Z. Yang, J. Wu, Y. Zhao, and Z. Zhou, “EdgeDuet: Tiling small object detection for edge assisted autonomous mobile vision,” in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.

[25] T.-W. Chin, R. Ding, and D. Marculescu, “AdaScale: Towards real-time video object detection using adaptive scaling,” in *Proc. Mach. Learn. Syst.*, 2019, pp. 431–441.

[26] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, “MoDNN: Local distributed mobile computing system for deep neural network.” in *Proc. Des. Automat. Test Eur. Conf. Exhib.*, 2017.

[27] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, “Adaptive parallel execution of deep neural networks on heterogeneous edge devices,” in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, 2019.

[28] F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. Van Gool, “Conditional probability models for deep image compression,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4394–4402.

[29] A. Khetan and Z. Karnin, “PruneNet: Channel pruning via global importance,” 2020, *arXiv: 2005.11282*.

[30] R. Xu et al., “Approxdet: Content and contention-aware approximate object detection for mobiles,” in *Proc. 18th Conf. Embedded Netw. Sensor Syst.*, 2020, pp. 449–462.

[31] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” 2015, *arXiv:1509.02971*.

[32] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016.

[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016.

[34] A. Howard et al., “Searching for MobileNetv3,” in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 1314–1324.

[35] J. Redmon and A. Farhadi, “YOLO9000: Better, faster, stronger,” 2016, *arXiv:1612.08242*.

[36] W. Liu et al., “SSD: Single shot multibox detector,” in *Proc. Eur. Conf. Comput. Vis.*, Springer, 2016, pp. 21–37.

[37] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.

[38] NVIDIA, “nvJPEG.” [Online]. Available: https://developer.nvidia.com/nvjpeg

[39] “Webcams,” 2023. [Online]. Available: https://www.webcamtaxi.com/

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*. Ieee, 2009, pp. 248–255.

[41] S. Che et al., “Rodinia: A benchmark suite for heterogeneous computing,” in *Proc. IEEE Int/Symp. Workload Characterization*, 2009, pp. 44–54.

[42] W. Li, D. Liu, K. Chen, K. Li, and H. Qi, “Hone: Mitigating stragglers in distributed stream processing with tuple scheduling,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2021–2034, Aug. 2021.

[43] W. Li, S. Chen, K. Li, H. Qi, R. Xu, and S. Zhang, “Efficient online scheduling for coflow-aware machine learning clusters,” *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2564–2579, Fourth Quart. 2022.

**Xueyu Hou** received the BS and MS degrees in electrical engineering from Xi’an Jiaotong University, and the PhD degree from the Electrical and Computer Engineering Department, New Jersey Institute of Technology (NJIT), advised by professor Tao Han. She was also a student in the Special Class of Gifted Young in Xi’an Jiaotong University. She is the recipient of the NJIT Hashimoto Prize 2024. Her current research interests include efficient artificial intelligence, human-centered computing, mobile edge computing, and sustainable computing.

**Yongjie Guan** received the BS degree in electrical engineering from the University of Electronic Science and Technology of China, and the master’s and PhD degrees from the Electrical and Computer Engineering Department, New Jersey Institute of Technology (NJIT), advised by professor Tao Han. His current research interests include mobile X reality system, mobile edge computing, unmanned aircraft systems, and human-centered computing.

**Nakjung Choi** (Senior Member, IEEE) received the BS and PhD degrees from the School of Computer Science and Engineering, Seoul National University, in 2002 and 2009, respectively. He is currently leading MNS (Mobile Network Systems) Department in NSSR (Network Systems and Security Research), Nokia Bell Labs, Murray Hill, USA, and DMTS (Distinguished MTS). Also, he has received several awards such as Best Paper Awards and Awards of Excellence. His research is on end-to-end network orchestration and automation, network control cross domains, 5G/5G-A/6G, dynamic network slicing, carrier-grade cloud-native, SDN/NFV, edge/fog computing & networking, and open radio access network (O-RAN).

**Tao Han** (Senior Member, IEEE) received the PhD degree in electrical engineering from the New Jersey Institute of Technology, in 2015. He is an associate professor with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology (NJIT). Before joining NJIT, He was an Assistant Professor with the Department of Electrical and Computer Engineering, University of North Carolina at Charlotte. He is the recipient of the NSF CAREER Award 2021, the Newark College of Engineering Outstanding Dissertation Award 2016, the NJIT Hashimoto Prize 2015, and the New Jersey Inventors Hall of Fame Graduate Student Award 2014. His papers win the IEEE International Conference on Communications (ICC) Best Paper Award 2019 and IEEE Communications Society's Transmission, Access, and Optical Systems (TAOS) Best Paper Award 2019. His research interests include mobile edge computing, machine learning, mobile X reality, 5G system, Internet of Things, and smart grid.