

CoEdge: Cooperative DNN Inference With Adaptive Workload Partitioning Over Heterogeneous Edge Devices

Liekang Zeng^{ID}, *Graduate Student Member, IEEE*, Xu Chen^{ID}, *Senior Member, IEEE*,
Zhi Zhou^{ID}, *Member, IEEE*, Lei Yang^{ID}, *Senior Member, IEEE*,
and Junshan Zhang^{ID}, *Fellow, IEEE*

Abstract—Recent advances in artificial intelligence have driven increasing intelligent applications at the network edge, such as smart home, smart factory, and smart city. To deploy computationally intensive Deep Neural Networks (DNNs) on resource-constrained edge devices, traditional approaches have relied on either offloading workload to the remote cloud or optimizing computation at the end device locally. However, the cloud-assisted approaches suffer from the unreliable and delay-significant wide-area network, and the local computing approaches are limited by the constrained computing capability. Towards high-performance edge intelligence, the cooperative execution mechanism offers a new paradigm, which has attracted growing research interest recently. In this paper, we propose CoEdge, a distributed DNN computing system that orchestrates cooperative DNN inference over heterogeneous edge devices. CoEdge utilizes available computation and communication resources at the edge and dynamically partitions the DNN inference workload adaptive to devices' computing capabilities and network conditions. Experimental evaluations based on a realistic prototype show that CoEdge outperforms status-quo approaches in saving energy with close inference latency, achieving up to 25.5% ~ 66.9% energy reduction for four widely-adopted CNN models.

Index Terms—Edge intelligence, cooperative DNN inference, distributed computing, energy efficiency.

I. INTRODUCTION

RECENT years have witnessed an ever-increasing number of Internet of Things (IoT) devices diving into

miscellaneous application domains, e.g., smart home [1], smart factory [2], autonomous driving [3], etc. This trend also drives the community to build smarter, faster, and greener intelligent applications at the network edge, pushing remarkable progress in smart healthcare, security inspection and disease detection [4]–[6]. Meanwhile, advances in Deep Neural Networks (DNNs) have shown unprecedented ability in learning abstract representation and extracting high-level features, promoting significant improvement in processing human-centric contents [7]. Motivated by this success, it is envisioned that employing DNNs to edge devices would enable and boost intelligent services, supporting brand new smart interactions between humans and their physical surroundings.

The essential demand of these services is to respond user's queries timely, e.g., recognizing voice commands [8], inspecting visitor's faces [9], and detecting heartbeat frequency [6], all within a matter of milliseconds. This also implies a soft-realtime requirement - if the result comes late, the user may turn to other applications, and the result can even be out of date and meaningless. Therefore, minimizing response latency and promising users' experience is of paramount importance. However, DNN-based applications are typically computation-intensive and resource-hungry [7], and service providers traditionally appeal to the resource-abundant cloud to satisfy the stringent responsiveness requirement [10]. Yet the Quality-of-Service (QoS) can still be poor and unsatisfactory due to the unreliable and delay-significant wide-area network connection between edge devices and the remote cloud [11], [12]. What's worse, for many smart applications with human in the loop, the sensory data can contain highly sensitive or private information. Offloading these data to the remote datacenter owned by curious commercial companies inevitably raises users' privacy concerns.

Intuitively, keeping data locally and processing tasks without external remote assists will preserve user privacy and avoid the remote network transmission. Unfortunately, local edge devices are generally with limited computing capability, making it hard to fulfill DNN execution under the latency Service-Level-Objective (SLO). For instance, if a smart home camera runs CNN-based face recognition to provide real-time inspection and warning, the response delay when running DNN locally may last for a few seconds, resulting in poor user experience and completely unusable service.

Manuscript received February 25, 2020; revised September 28, 2020; accepted November 18, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Tang. Date of publication December 16, 2020; date of current version April 16, 2021. This work was supported in part by the National Science Foundation of China under Grant U20A20159, Grant U1711265, and Grant 61972432; in part by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2017ZT07X355; in part by the Pearl River Talent Recruitment Program under Grant 2017GC010465; and in part by the U.S. National Science Foundation under Grant IIS-1838024, Grant EEC-1801727, and Grant CNS-1950485. (Corresponding author: Xu Chen.)

Liekang Zeng, Xu Chen, and Zhi Zhou are with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China (e-mail: zenglk3@mail2.sysu.edu.cn; chenxu35@mail.sysu.edu.cn; zhouzhi9@mail.sysu.edu.cn).

Lei Yang is with the Department of Computer Science and Engineering, University of Nevada, Reno, NV 89557 USA (e-mail: leiy@unr.edu).

Junshan Zhang is with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85287 USA (e-mail: junshan.zhang@asu.edu).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TNET.2020.3042320>, provided by the authors.

Digital Object Identifier 10.1109/TNET.2020.3042320

1558-2566 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.

See <https://www.ieee.org/publications/rights/index.html> for more information.

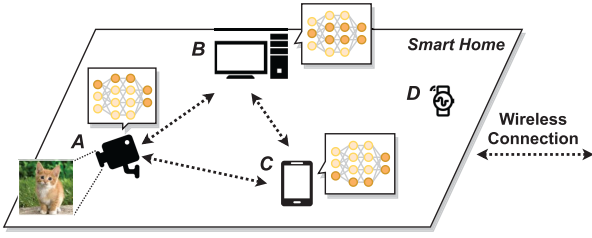


Fig. 1. An example of cooperative DNN inference in a smart home scenario. As the raw image is captured, device A decides a cooperative execution plan and distributes the workload to devices B and C. According to the plan, the devices perform cooperative inference in response to the DNN task.

To tackle these challenges, a promising approach is to exploit available computation resources in the proximity to the data source with the emerging edge intelligence paradigm [13]. Instead of uploading data to the remote cloud or keeping all computation at the single local device, edge intelligence enjoys real-time response as well as privacy preservation by offloading computing workload within a manageable range. As Fig. 1 illustrates, we can utilize the diverse computing resources in a smart home (with inspection camera, smartphone, tablet, and desktop PC) to accelerate the CNN-based face recognition. Specifically, the source device A can distribute the inference workload to devices B and C, and perform cooperative inference via high-bandwidth local wireless connection (e.g., WiFi). Nevertheless, this paradigm brings some key challenges to be addressed: (1) how to decide the workload assignment tailored to the resource heterogeneity of edge devices, (2) how to optimize the system performance with the presence of network dynamics, and (3) how to orchestrate computation and communication during cooperative inference runtime.

To answer these questions, we propose CoEdge (**Co**operative **E**dge), a runtime system that orchestrates cooperative DNN inference over multiple heterogeneous edge devices. CoEdge does not apply any structural modifications or tuning requirements to the given DNN model, and does not sacrifice model accuracy as it reserves input data and model parameters of the given DNN model. CoEdge employs a similar parallel workflow as DeepThings [14], where the input is split initially and the execution is parallelized on multiple devices at runtime. While DeepThings leverages a layer fusion technique to reduce communication overhead, CoEdge proposes to optimize workload allocation to maximally utilize heterogeneous edge resources. By optimizing the computation-communication tradeoff, CoEdge optimally partitions the input inference workload, where the partitions' sizes are chosen to match devices' computing capabilities and network conditions to improve system performance in both latency and energy metrics. We implement CoEdge using a realistic prototype with Raspberry Pi 3, Jetson TX2, and desktop PC. Experimental evaluations show $7.21\times \sim 4.49\times$ latency speedup over the local approach and up to 25.5% $\sim 66.9\%$ energy saving comparing with existing approaches for four popular DNN models.

In summary, this paper makes the following contributions.

- We propose CoEdge, a distributed DNN computing system that orchestrates cooperative inference over heterogeneous devices to minimize system energy consumption while promising response latency requirement.
- We identify the impacts of workload partitioning on cooperative inference workflow, and build a constrained programming model on workload distribution optimization. We prove the NP-hardness of the problem, and devise a fast approximated algorithm to decide the efficient partitioning policy in real-time tailored to devices' diverse computing capabilities and network conditions.
- We implement a multi-device prototype using heterogeneous edge devices, and evaluate CoEdge on four widely-adopted DNN models to corroborate its superior performance.

The rest of this paper is organized as follows. Section II briefly reviews background on DNN inference, and discusses opportunities and challenges based on a case of cooperative inference. Section III presents CoEdge design and its workflow. Section IV builds the system model and describes our workload partitioning algorithm. We explain implementation details in Section V and evaluate the prototype in Section VI. Section VII provides related works. Section VIII discusses limitation and extension of CoEdge, and Section IX concludes. The appendix (in the supplementary material) details the proofs of Theorem 1 and 2.

II. BACKGROUND AND MOTIVATION

In this section, we briefly review conventional CNN inference and cooperative inference. We study a case of cooperative inference and discuss potential challenges behind that.

A. Deep Neural Network Inference

In this paper, we focus on the classical Convolutional Neural Networks (CNNs) as they are widely adopted across a board spectrum of intelligent services, including image classification, object detection, and semantic segmentation, etc.

Fig. 2 depicts a conventional CNN inference for image classification task from a perspective of feature maps. As we can see, a conventional CNN inference can be viewed as a series of successive algorithmic operations on feature maps. These operations¹ comprise of convolution, pooling, batch normalization, activation, and fully-connected computation, etc. In light of the functionality of the operations, the inference process can be separated into two stages. The first stage is the feature extraction stage, where the model processes every pixel in the input image to generate hidden feature representations. Following that, at the second stage, these features are classified by the fully-connected layers, exporting results in a probabilistic form.

B. Case Study: Cooperative Inference With Two Devices

The key impediment of deploying CNN at the network edge lies in the gap between intensive CNN inference computation and the limited computing capability of edge devices.

¹For ease of illustration, only some of the operations are drawn in Fig. 2

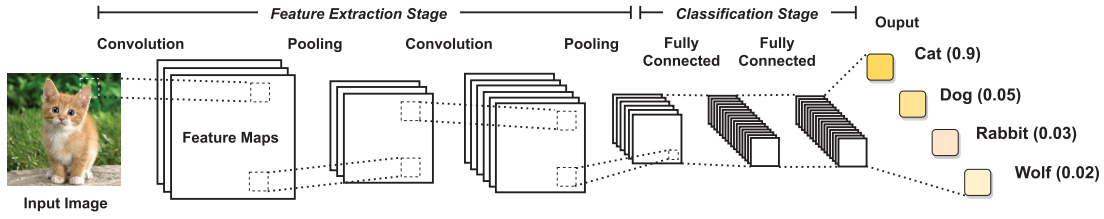


Fig. 2. Conventional CNN inference workflow, which is typically in two stages. In the first stage, CNN processes the input image to extract hidden features through operations like convolution and pooling, and generates multidimensional feature maps. In the second stage, CNN classifies the feature maps by fully-connected layers and obtains the inference result.

TABLE I
RASPBERRY PI 3 SPECIFICATIONS [15]

Hardware	Specifications	
CPU	1.2GHz Quad Core ARM Cortex-A53	
Memory	1GB LPDDR2 900MHz	
GPU	No GPU	
Power	Idle	1.3W
	Fully Loaded	6.5W
	Average Observed	3W

TABLE II
JETSON TX2 SPECIFICATIONS [16]

Hardware	Specifications	
CPU	2.0GHz Dual Denver 2 + 2.0GHz Quad Core ARM Cortex-A57	
Memory	8GB LPDDR4 1.6GHz	
GPU	Pascal Architecture 256 CUDA Core	
Power	Idle	5W
	Fully Loaded	15W
	Average Observed	9.5W

To bridge this gap, we can utilize the cooperative inference mechanism to exploit available computing resources at the edge. A straightforward solution of that, for example, is the master-worker paradigm that offloads inference workload to external infrastructure. To obtain a better understanding of cooperative inference, we use a real hardware testbed to emulate this solution.

As a case study, we employ a Raspberry Pi 3 and a Jetson TX2, on behalf of weak IoT devices and mobile AI platforms at the edge, respectively. Table I and II presents their specifications and reported power parameters, which are measured with Monsoon High Voltage Monitor [17] using the methodology in [18]. For each inference task, we input one single image to the Pi and then offloads a part of the image to the Jetson. The two devices parallelize the DNN execution and their results are finally aggregated to the Pi as output. We measure the end-to-end latency of this process, i.e., from the image input to the inference result output; and we record the average latency of fulfilling the inference task over 100 runs. We implement AlexNet [19] with TensorFlow Lite [20] on both devices, and run the model with the same image from ImageNet [21]. For the bandwidth between two devices, we fix it at 1MB/s using the traffic control tool tc [22].

We define *offloading ratio* to indicate how much data is offloaded from the Raspberry Pi to the Jetson TX2. For instance, when the ratio is 0.5, we split the input image

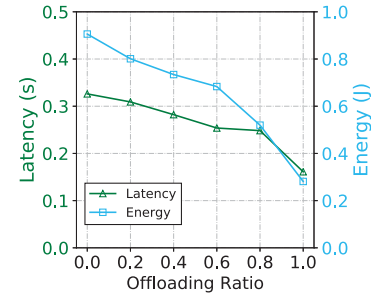


Fig. 3. The total latency and energy consumption under varying offloading ratio, i.e., the proportion of data that is offloaded from the Raspberry Pi 3 to the Jetson TX2.

along the height into two equal parts, and transfer one of them to the Jetson TX2. In particular, a zero ratio indicates performing inference at the Raspberry Pi locally, while the ratio equals to 1.0 if offloading all workload to the Jetson TX2. Fig. 3 shows the latency and energy overheads under varying offloading ratio. Through this experiment, we derive the following observations.

- Jetson TX2 enjoys better performance than Raspberry Pi 3. When the ratio is zero, the system consumption is only the computation cost of Pi, while at the 1.0 case, the total cost comprise of the input offloading overhead, the DNN computation overhead in Jetson and the overhead for transferring result back. However, the former still takes higher cost than the latter in terms of both latency and energy. Note that fully offloading workload to the Jetson (i.e., *offloading ratio* = 1) may not necessarily yield the lowest costs if the network fluctuates.
- Cooperative inference is more economical than local inference given the favourable network condition. As the offloading ratio increases, both latency and energy drop. In other words, the system cost decreases via harvesting the cooperators' computing resource.
- The curve of latency drops faster as the offloading ratio increases. This is because the DNN execution is parallelized in cooperative inference and the end-to-end inference latency is straggled by the slower one. Therefore, in high bandwidth environments, assigning more workload to Jetson TX2 benefits performance improvement better.

C. Merits and Challenges

We see that, from the above observations, cooperative mechanism has potential to improve inference performance

with multiple devices, which are exactly what edge scenarios possess. More precisely, we envision the deployment of the cooperative inference system in an environment such as smart home or smart factory, wherein the devices are managed by the same owner and thus they are willing to cooperate and share their resources. This brings several major merits as well as challenges.

Merits. On the one hand, comparing with local inference, cooperative inference has significant potential in reducing latency and energy costs via harvesting idle computing resources at the edge. On the other hand, other than the cloud approach that uploads data to the remote datacenter, the cooperative approach keeps data within user's control scope, therefore avoiding the delay-significant wide-area network as well as privacy issues.

Challenges. To effectively exploit computing resources at the edge, we need to felicitously factor the computing capabilities of edge devices considering magnitude and heterogeneity. Also, given the dynamic network inherently in edge computing, an efficient workload allocation solution that jointly considers systematic costs is desired. More specifically, it is crucial to decide which device to participate in the cooperative inference and how much workload each device affords. Besides, since the cooperative mechanism parallelizes CNN inference in a distributed manner, the system needs to orchestrate the computation and communication over multiple devices.

We address these challenges by designing a cooperative system, CoEdge, through orchestrating the available resources from heterogeneous edge devices.

III. COEDGE DESIGN AND WORKFLOW

In this section, we present CoEdge design and the workflow of cooperative CNN inference. We further explore how workload partitioning impacts parallel processing in terms of computation and communication.

A. CoEdge Design

For ease of illustration, we differentiate between devices on their roles in the cooperation. For the device that launches a CNN inference task, we label it as the *master device*, and for the device that joins the cooperation, it is marked as the *worker device*. The master device is responsible for registering participated devices, generating a feasible workload partitioning plan, and managing the cooperative inference over worker devices. Note that a device can be the master and the worker at the same time since it can retain CNN workload in situ.

Fig. 4 illustrates the architecture overview of CoEdge, which works in two phases, namely the setup phase and the runtime phase. In the *setup* phase, CoEdge records the execution profiles of each device. In the *runtime* phase, CoEdge creates a cooperative inference plan that determines the workload partitions and their corresponding assignment, using the profiling results collected in the setup phase and the network status. According to the cooperation plan, the master

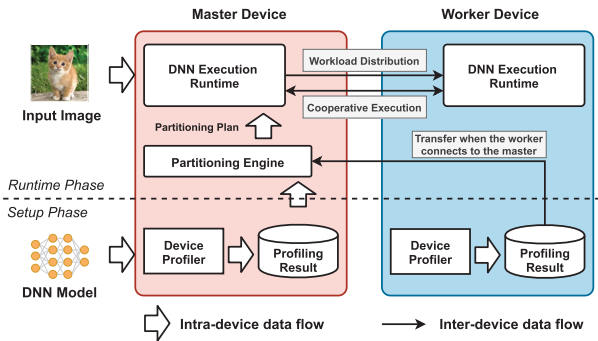


Fig. 4. CoEdge architecture overview, which works in two phases. In the setup phase, the devices profile parameters to sketch their computing capabilities information. In the runtime phase, the master device creates a partitioning plan using the collected profiling results. According to the plan, the master device distributes the workload and performs cooperative inference with worker devices.

distributes the workload partitions to the workers and then performs cooperative execution collaboratively.

Setup phase. Whenever a CNN-based application is installed, *Device Profiler* runs the CNN models locally and records *Profiling Results*. These results sketch the device's computing capability, including the computation intensity, computation frequency and power parameters, which will be detailed in Section IV-A.

Runtime phase. The runtime phase starts when the master raises a CNN inference query. As the image inputs, the master establishes connections with worker devices and pulls their profiling results. Since the size of the profiling results is very small (tens of bytes in our prototype), the transmission overhead for transferring them is negligible. As the master receives the profiling data, the *partitioning engine* in the master device generates a workload allocation plan using the adaptive workload partitioning algorithm (explained in Section IV-C). According to the plan, *DNN execution runtime* distributes the workload partitions to workers and performs cooperative inference in response to the query.

B. Cooperative Inference Workflow

In this work, we exploit model parallelism to partition CNN inference over multiple devices. Under model parallelism, CNN model parameters are divided into subsets and assigned to multiple edge nodes. With respective parameters, each device accepts a necessary part of the input feature maps and generates a portion of the output feature maps. Concatenating all these portions yields the complete output of each layer.

Fig. 5 provides an instance of cooperative inference workflow with three devices from a perspective of feature maps. The cooperative inference begins when the image is piecewise split into partitions. Note that to accommodate devices' heterogeneity, the partition sizes are differentiated to match device capabilities. The partitions are then distributed from the master to three devices (i.e., devices A, B, and C in Fig. 5). At the feature extraction stage, the three devices execute their workload in parallel, while at the classification stage, their execution results are aggregated to one of them (device B in Fig. 5). This aggregation is to avoid excess communication

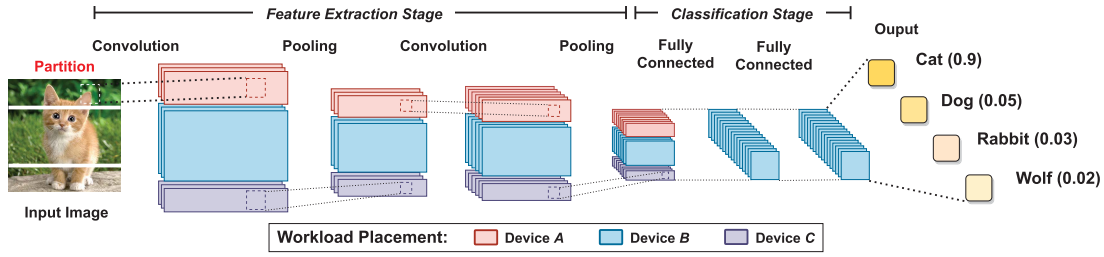


Fig. 5. Cooperative CNN inference workflow of CoEdge. The input image is piece-wise partitioned to patches before execution. In feature extraction stage, these patches are distributed to devices A, B and C, respectively, and then in classification stage, the feature map fragments are aggregated to finish the remaining execution.

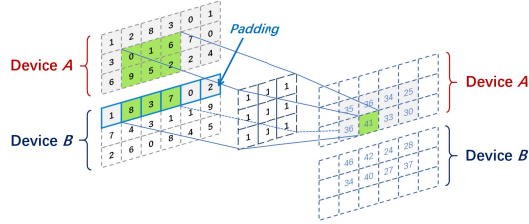


Fig. 6. Example of a convolution operation for cooperative inference. The input feature map partitions with each of 3×6 size locate at devices A and B, respectively. To generate the output feature map through the 3×3 convolution kernel, device A needs to pull the padding data of 1×6 size from device B.

overhead caused by the nature of fully-connected computation, which requires repeating data access on the feature vectors.

Generalization. Based on the workflow in Fig. 5, it is feasible to accommodate various CNNs with complex structures by redesigning some details. For example, for CNNs without fully-connected layers (e.g., Network in Network [23]), we can reduce the classification stage in Fig. 5. To adapt CNNs with skip connections (e.g., ResNet [24]), we can keep intermediate output results on each device at the shortcut starting point and release these data at the shortcut destination to collect the data when needed.

C. Impact of Workload Partitioning

The way of piece-wise partitioning significantly affects the communication between devices, especially for convolution operations that process data across partition boundaries. For instance, Fig. 6 shows a typical convolution operation with two partitions. To compute convolution over the 3×6 partition with the 3×3 kernel, device A needs to fetch the 1×6 margin row in device B's partition. In general, for the kernel whose size k is greater than 1, each device needs to pull the padding data of $\lfloor k/2 \rfloor$ size along the split dimension from the neighboring device. In some extreme cases, when the kernel size is very large but the neighboring partition size is very small, the padding range may even across three or more devices, which could incur extravagant communication overhead.

To reduce the communication between devices, some prior works [14], [25] exploit sending redundant data in advance to avoid the padding issue. However, while transferring redundant data takes additional communication cost, preparing necessary data beforehand for a number of CNN layers incurs extra storage overhead. In this work, we address the padding issue

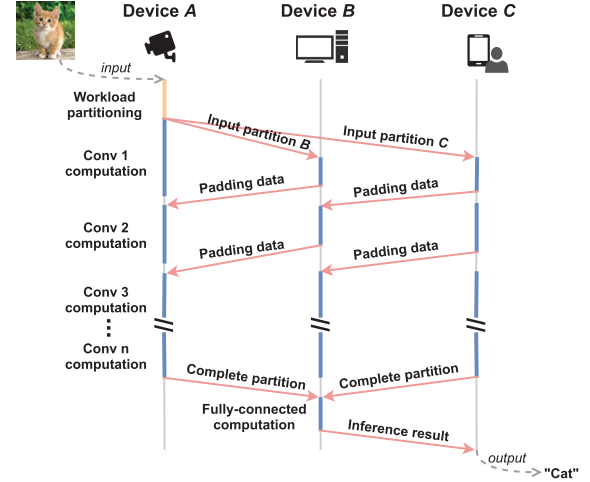


Fig. 7. The communication pattern of a CoEdge runtime instance with three devices. For convolution computation, each device pulls the necessary padding data from its neighboring device. For fully-connected computation, the feature map partitions are aggregated. The final inference result is transferred to a user-specified location, i.e., device C in the figure.

by imposing a principle that requires the allocated partition size in the neighboring device to be not smaller than the padding size, unless it owns no partition. This principle ensures that the padding data can be always acquired from only the neighboring device as long as it has data. That is, the transmission of the padding data merely happens once, and thus we reduce the overhead in establishing additional connections. To illustrate that, Fig. 7 show the communication pattern of the example in Fig. 5. Initially, the input image is partitioned and distributed to corresponding devices, along with the padding data for the first convolution layer. For the following layers, each device only connects to its neighbor device and fetches padding data for convolution computation. This pattern holds until all convolutions are completed, and then the separated feature map partitions are aggregated to one of the devices for fully-connected computation. The inference result is finally returned to a user-specified device (device C for example in Fig. 7).

Under this principle, finding an appropriate workload assignment matters significantly for system performance. For instance, offloading a large portion of workload to a device that owns high bandwidth but poor computing capability may not lead to a lower execution latency. To deploy cooperative inference optimally, we need to match the assigned partition

size to the computation and communication resource of each device. We achieve this goal by designing a workload partitioning algorithm that is adaptive to the computing capabilities of available devices and the dynamic network status.

IV. ADAPTIVE WORKLOAD PARTITIONING

The objective of optimizing workload allocation is to improve cooperative inference performance in both latency and energy metrics. For the simplicity of problem definition, we target at meeting the latency requirement while minimizing the energy costs. Assuming an execution deadline D , the optimization problem is to optimally allocate the workload so that the system energy consumption is minimized while promising the execution deadline D , given the available computation and communication resources.

In the following, we present a detailed formulation of this problem and our workload partitioning algorithm.

A. Problem Formulation

We assume that the devices are available and relatively stable during the inference runtime. This can be relevant as executing an inference task is typically in a period of seconds, and many edge environments are maintained statically in independent spaces, such as smart home and smart factory. Besides, the underlying support of intelligent services in such scenarios usually employs a few commonly-adopted DNN models and frequently run similar types of DNN inference tasks. Therefore, we suppose that the DNN models have been loaded ahead of inference queries, and can be used to compute input tensors as soon as necessary data are prepared.

Since a CNN model typically encompasses many layers, we model the cooperative inference process from single-layer to multi-layer, progressively. The single-layer formulation focuses on sketching the workload partitioning constraints and shaping the performance of single-layer, while the multi-layer part aims at summarizing the system behavior for the whole workflow. Prior to that, we define the necessary concepts and notations as follows:

- A layer l is an algorithmic operation in a CNN model. In our formulation, a layer refers to either a convolution (Conv) or a fully-connected (FC) layer. Given a CNN model, $\mathcal{L} = [1, 2, \dots, L]$ denotes the layers in order.
- A partitioning solution π is a group of coterminous partitions of the input image, which is generated by piece-wise partitioning along one dimension. For the input partition assigned to device i , a_i represents the number of rows that it covers. Hence, given the devices' indices $\mathcal{N} = [1, 2, \dots, N]$, $\pi = [a_1, a_2, \dots, a_N]$. We denote the workload as the input feature map partition to be processed on each DNN layer. For layer l , the workload size of the i -th partition is r_{li} , which can be obtained by calculating the partition's data size.
- A configuration tuple $(k, c_{in}, c_{out}, s, p)_{li}$ denotes the l -th layer's computation task on the i -th partition, which is characterized by the layer's configuration, i.e., convolution kernel size k , input channels c_{in} , output channels c_{out} , stride s , and padding p . This tuple is applicable for both

convolution and FC layers since FC computation can be viewed as a special case.² In particular, as discussed in Section III-C, the padding size of convolutional layers is supposed to be smaller than the size of the partition on the last neighboring device, unless it owns no data. We formulate this principle as Eq. (1), where $\mathbb{1}_{\{a_i > 0\}}$ is an indicative function that values 1 if $a_i > 0$ or else 0. This constraint is essentially equivalent to the disjunction of $a_i \geq p_{i+1}$ and $a_i = 0$.

- A resource tuple $(\rho, f, m, P^c, P^x)_i$ specifies the resource profile of device i . Here, ρ is defined as the computing intensity (in processing cycles per 1KB input) of the given DNN model, which is measured by application-driven offline profiling [26] in the setup phase. f is the device's CPU frequency, reflecting its computing capability in a coarse granularity. m is the available maximum memory capacity for inference tasks. For a single device that only processes CNN workloads, m is the volume of memory excluding the space taken by the underlying system services, e.g., I/O services, compiler, etc. P^c and P^x denote the computation power and the wireless transmission power, respectively.

1) *Single-Layer Formulation*: There are some numerical constraints on the partition sizes. Eq. (1) imposes the size restriction with the padding size as discussed in Section III-C, and Eq. (2) claims a_i is a nonnegative integer. Eq. (3) presents that the concatenation size of all partitions along the height dimension equals to this dimension's size H . The piece-wise partitioning can be conducted along either the height H or width W of the input. In our experiments, we split along the height H without loss of generality.

$$a_i \geq p_{i+1} \mathbb{1}_{\{a_i > 0\}}, \quad i \in \mathcal{N}, \quad (1)$$

$$a_i \geq 0, a_i \in \mathbb{Z}, \quad i \in \mathcal{N}, \quad (2)$$

$$\sum_{i \in \mathcal{N}} a_i = H. \quad (3)$$

The workload size r_{li} of a partition is constrained by the device's available memory capacity m_i , as in Eq. (4). Here, we only limits the memory footprint on the size of per-layer inputs for the sake of simplicity, while the runtime memory may not be exactly r_{li} . For practical deployment cases, emerging techniques on characterizing the detailed CNN execution memory (e.g., [27]) can be adopted, and the deep learning platform-related memory footprint can be added into the left-hand side of Eq. (4) as an enhancement.

$$r_{li} \leq m_i, \quad i \in \mathcal{N}, \quad l \in \mathcal{L}. \quad (4)$$

During a single layer's execution, the system takes time and energy on two aspects, computation and communication. For computation, we calculate the latency and energy by first approximating the computing cycles of given partitions. As demonstrated in previous empirical studies [26], [28], [29], for many data processing tasks as exemplified by data encoding and decoding, the required computing cycles are proportional to their input data sizes. This means that, a constant

²This tuple depicts a fully-connected computation when the input feature map's size is $1 \times 1 \times c_{in}$, the output feature map's size is $1 \times 1 \times c_{out}$, kernel size $k = 1$, stride $s = 1$, and padding $p = 0$.

computing intensity (in computing cycles per unit data) exists for such tasks, and we can use it to capture the effective computing capability of a specific device. Existing literature, such as [30]–[32], has leveraged this observation to characterize deep learning workloads, and in this work, we adopt it to estimate the computing cycles amount given the partitions and DNN layers. Concretely, in Eq. (5), we assess the total processing cycles of the i -th partition by multiplying the device's computing intensity ρ_i with the workload size r_{li} . Moreover, for each respective DNN layer, CNN inference typically conducts a feed-forward execution without any branch operation or recurrent computation [7], indicating that its execution latency is approximately linear to the computing cycles. Therefore, the latency T_{li}^c for computing layer l is then appraised via dividing the total processing cycles by the computation frequency f_i , and the energy is the product of T_{li}^c and the computation power P_i^c in Eq. (6). Note that Eq. (6) only reckons on dynamic energy. Static energy consumption, e.g., those for maintaining basic system-level services, are not considered in our formulation.

$$T_{li}^c = \frac{\rho_i r_{li}}{f_i}, \quad i \in \mathcal{N}, l \in \mathcal{L}, \quad (5)$$

$$E_{li}^c = P_i^c T_{li}^c, \quad i \in \mathcal{N}, l \in \mathcal{L}. \quad (6)$$

For communication, let $b_{i,j}$ be the available bandwidth between devices i and j . Particularly, $j = i$ implies delivering data from a device to itself, and $b_{i,i}$ is the memory bandwidth. In our experiment, $b_{i,i}$ is set as 12.8GB/s by default, which is the typical memory bandwidth of DDR3 [33]. Initially, the communication occurs when the master device (noted as device **M**) distributes input partitions to worker devices, the transmission time is therefore calculated in the $l = 1$ case of Eq. (7). For communication of pulling the padding data from the neighboring device, its transmission time is described as the $l > 1$ case. For the sake of simplicity, Eq. (7) does not take the queuing delays into account since we are optimizing inference for respective single image input. Streaming input, in which case the queuing delays significantly matter, are left for future work. With the transmission power P_i^x , we acquire the dynamic energy of communicating with device i on layer l in Eq. (8).

$$T_{li}^x = \begin{cases} \frac{a_i}{b_{\mathbf{M},i}}, & l = 1, i \in \mathcal{N}, \\ \frac{b_{\mathbf{M},i}}{b_{i,i+1}}, & l > 1, l \in \mathcal{L}, i \in \mathcal{N}, \end{cases} \quad (7)$$

$$E_{li}^x = P_i^x T_{li}^x, \quad i \in \mathcal{N}, l \in \mathcal{L}. \quad (8)$$

2) *Multi-Layer Formulation*: The key challenge of extending the formulation from a single layer to multiple layers lies in the synchronization mechanism during parallel processing. Fig. 8 presents a job breakdown of a CoEdge instance - processing one image over three devices. As we can see, each device processes computation and communication jobs alternately, and they trigger synchronization periodically whenever a communication job (except for the initial communication job) is accomplished. The contents of the synchronizations are the requisite padding data for convolutional computation. During the interval between two synchronizations, there is no

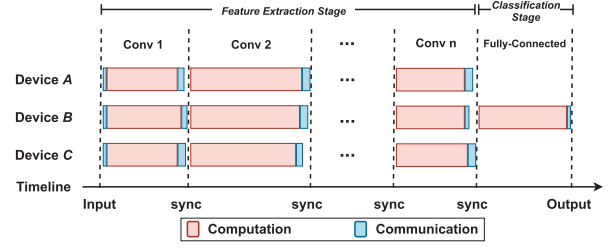


Fig. 8. The job breakdown of a CoEdge runtime instance with three devices. Each device processes computation and communication jobs alternately, and the system performs in a Bulk Synchronous Parallel (BSP) mechanism.

data dependency between devices, and thus they process jobs in parallel. These scattered feature map partitions are finally aggregated at the classification stage for FC computation. Hence, the whole process works in a Bulk Synchronous Parallel (BSP) mechanism [34].

To summarize the cost of the whole process, we denote E^c and E^x as the total energy consumption of computation and communication, respectively, which are obtained by summing up the energy of all devices for all layers in Eq. (9) and (10). We count the total physical latency T according to the BSP model and obtain Eq. (11). Concretely, we acquire T by calculating the maximum latency of all devices and then summing up the physical latency of all intervals. It is worth noting that Eq. (11) has counted the latency of FC layers, as the maximum latency of all device is essentially that of the selected device in the classification stage.

$$E^c = \sum_{l \in \mathcal{L}} \sum_{i \in \mathcal{N}} E_{li}^c, \quad (9)$$

$$E^x = \sum_{l \in \mathcal{L}} \sum_{i \in \mathcal{N}} E_{li}^x, \quad (10)$$

$$T = \sum_{l \in \mathcal{L}} \max_{i \in \mathcal{N}} (T_{li}^c + T_{li}^x). \quad (11)$$

Given the execution deadline D , the targeted problem is to decide an optimal partitioning solution $\pi = [a_1, a_2, \dots, a_N]$ with the objective of minimizing total energy without violating the execution deadline D . Hence, we can formulate the cooperative inference optimization as the following problem.

$$\begin{aligned} \mathcal{P1}: \quad & \min. E^c + E^x \\ & \text{s.t. } T \leq D, \\ & (1), (2), (3), (4). \end{aligned}$$

Theorem 1: Problem $\mathcal{P1}$ is an NP-hard problem.

We prove Theorem 1 by identifying $\mathcal{P1}$ as an Integer Linear Programming problem (detailed in Appendix A). In a typical smart factory deployment, there may be tens or even hundreds of edge devices, indicating that the decision space of $\mathcal{P1}$ can be huge (which grows exponentially with the increase of edge devices' amount) according to Theorem 1. Therefore, to generate a solution in real-time, it is necessary to find an efficient solving method to $\mathcal{P1}$.

B. Problem Transformation

A Linear Programming (LP) problem is a kind of optimization towards a linear objective function subject to linear

equality or inequality constraints, and the Integer Linear Programming (ILP) problem is a special case where all optimization variables are integers [35]. As proved in Appendix A, $\mathcal{P}1$ is an ILP. The difficulty of solving $\mathcal{P}1$ lies in the discreteness of integer variable a_i . To produce a feasible solution to $\mathcal{P}1$ efficiently, we relax $\mathcal{P}1$ by introducing a continuous variable λ_i to approximate a_i . Eq. (12) defines the relation between λ_i and a_i , where H is the input's height and λ_i describes the proportion that the i -th partition covers. Since the input of CNN inference are usually of a large size (e.g., typically of 224×224 size from ImageNet [21] dataset), the approximation error is tiny and tolerated. Eq. (13), (14), and (15) show the numerical constraints for λ_i , which are derived from Eq. (1), (2), and (3), respectively.

$$a_i = \lambda_i H, \quad i \in \mathcal{N}, \quad (12)$$

$$\lambda_i H \geq p_{i+1} \mathbb{1}_{\{\lambda_i > 0\}}, \quad i \in \mathcal{N}, \quad (13)$$

$$\lambda_i \geq 0, \quad i \in \mathcal{N}, \quad (14)$$

$$\sum_{i \in \mathcal{N}} \lambda_i = 1. \quad (15)$$

Eq. (13) is essentially equivalent to the expression of $\lambda_i H \geq p_{i+1}$ or $\lambda_i = 0$. Since $\lambda_i = 0$ is a potential solution, it is feasible to separate solving λ_i 's value and checking whether $\lambda_i \geq p_{i+1}$ to two steps. Therefore, we relax the constraint Eq. (13) as $\lambda_i H \geq 0$, i.e., $\lambda_i \geq 0$, and $\mathcal{P}1$ can be transformed into the following problem $\mathcal{P}2$.

$$\begin{aligned} \mathcal{P}2: \quad & \min. E^c + E^x \\ & \text{s.t. } T \leq D, \\ & (4), (14), (15). \end{aligned}$$

$\mathcal{P}2$ is fundamentally a special case of $\mathcal{P}1$. Particularly, on the solution to $\mathcal{P}2$, there may be some devices that are assigned with tiny workload ($\exists i \in \mathcal{N}, 0 \leq \lambda_i < p_{i+1}$), while on the solution to $\mathcal{P}1$, the workload size on all devices must be larger than or equal to the padding data size unless it is zero ($\forall i \in \mathcal{N}, \lambda_i \geq p_{i+1}$ or $\lambda_i = 0$). Regardless of the potential solution $\lambda_i = 0$ to problem $\mathcal{P}1$, the main difference between $\mathcal{P}1$ and $\mathcal{P}2$ is the setting of threshold, i.e., $\mathcal{P}1$ sets the threshold as p_{i+1} while $\mathcal{P}2$ sets 0. Hence, we can exploit $\mathcal{P}2$'s solution to iteratively approach $\mathcal{P}1$'s solution by checking whether it satisfies the threshold constraint Eq. (13).

Theorem 2: Problem $\mathcal{P}2$ is a Linear Programming problem.

Theorem 2 (proved in Appendix B) reveals that $\mathcal{P}2$ is a LP, which can be efficiently solved by existing mature programming solvers (e.g., CPLEX [36]). By them, it is feasible to fast approximate the solution to $\mathcal{P}1$.

C. Workload Partitioning Algorithm Design

We propose a threshold-based workload partition algorithm for $\mathcal{P}1$ using existing programming solvers, as presented in Algorithm 1. The key idea of Algorithm 1 is to gradually narrow down the selection of participating devices (i.e., the devices that are assigned with workload) by checking the threshold constraint and iteratively approach the solution.

The input of Algorithm 1 includes CNN layer configurations, available computation and communication resources,

Algorithm 1 Workload Partitioning Algorithm

Input:

\mathcal{N} : Available devices $[1, 2, \dots, N]$
 \mathcal{L} : CNN layers $[1, 2, \dots, L]$
 $(k, c_{in}, c_{out}, s, p)_{li}, \forall i \in \mathcal{N}, \forall l \in \mathcal{L}$: Configuration tuples
 $(\rho, f, m, P^c, P^x)_i, \forall i \in \mathcal{N}$: Resources tuples
 $b_{i,j}, \forall i, j \in \mathcal{N}$: Bandwidths
 D : Execution deadline

Output:

π : Assigned workload proportions $[\lambda_1, \lambda_2, \dots, \lambda_N]$

```

1: Procedure PARTITION( $\mathcal{N}$ )
2:   if  $\mathcal{N}$  is empty then
3:     return NULL ▷ no feasible solution
4:   Solve  $\pi$  from  $\mathcal{P}2$ 
5:   if  $\pi$  satisfy Eq. (13) then
6:     return  $\pi$ 
7:   else
8:     Find the index set  $\mathcal{N}_0$  of zero elements in  $\pi$ 
9:     Find the minimum element  $\lambda_m$  in  $\pi$ 
10:     $\mathcal{N} \leftarrow \mathcal{N} - \mathcal{N}_0 - \{m\}$ 
11:    return PARTITION( $\mathcal{N}$ )
12: end Procedure

```

and the execution deadline. These inputs provide parameters for $\mathcal{P}1$ and $\mathcal{P}2$. The output is the workload partitioning solution to $\mathcal{P}1$.

Algorithm 1 begins with checking whether \mathcal{N} is empty - if \mathcal{N} is an empty set, there is no available devices to perform cooperative inference, and thus no feasible solution to $\mathcal{P}1$. Otherwise, we solve π from $\mathcal{P}2$ with a LP solver. Then we check whether the obtained π satisfy Eq. (13), the threshold constraint of $\mathcal{P}1$. If so, the current version of π is a feasible solution and is immediately returned. Or else, there must be some elements in π that are smaller than the required padding size. In this case, we remove part of these unsatisfied elements from the available devices list: firstly we remove zero elements since the zero workload assignment indicates that the device would not participate in cooperative inference; next we find the minimum from the rest elements in π and remove it from \mathcal{N} . After that, the algorithm goes to the next recursion to acquire the new partition solution with the updated \mathcal{N} and checks the result for $\mathcal{P}1$ again. The recursion continues until it finds a feasible solution - if any - or a null flag as the unfeasible signal (this could happen when the deadline constraint is too strict to satisfy).

The programming solver for our LP problem runs efficiently. In our experiment we use CPLEX and the runtime overhead is smaller than 1ms. Since the total recursion times in Algorithm 1 will not exceed N (the total number of available devices), the solving process of Algorithm 1 is very fast (<10 ms) and will not cause side-effect on the pursuit of latency SLO.

V. PROTOTYPE IMPLEMENTATION

We employ TensorFlow Lite [20] as the backend engine to execute CNN layers, and implement the communication

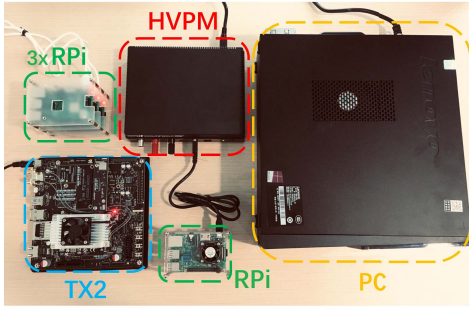


Fig. 9. Our experimental prototype uses four Raspberry Pis (RPI), one Jetson TX2 and one desktop PC. Their specifications are listed in Table I, II and III. We employ the Monsoon High Voltage Power Monitor (HVPM) to measure the energy.

module based on gRPC [37]. In the following, we provide the implementation details of CoEdge.

Deployment and profiling. Since any one of the devices in the environment may launch a CNN inference task, the employed CNN models are trained and installed on all devices in advance. As the model is installed, we use TensorFlow benchmark tool to profile the latency of one inference and measure the energy with the Monsoon High Voltage Power Monitor [17]. For each CNN model, we run it for once as warm-up and then record the execution time with 50 runs without break. The aim of warm-up running is to alleviate the impact of weight loading and TensorFlow initiation since we have omitted these overheads in the formulation. The execution tasks on all devices are the same - perform CNN inference on the same image from ImageNet [21]. We take the mean values as the measuring results and derive the resource tuple parameters based on them.

The computation frequency f is directly from known specifications. With f and the measured latency, we can estimate the total computing cycles of one inference. Dividing the cycles' amount to the processed image size yields the computing intensity ρ . We obtain the memory capacity m by observing the available memory space of an idle system. For power parameters P^c and P^x , we measure them by calculating the measured computation/communication energy and delay.

Workload partitioning and distribution. To create the workload allocation plan efficiently, We run the workload partitioning algorithm based on IBM ILOG CPLEX [36], a linear programming solver package. If the algorithm returns a feasible solution, we segment the input image accordingly and send the partitions to the corresponding devices. Otherwise, the algorithm returns an infeasible signal, which means the deadline is set too strict. In this case, we choose to offload all workload to the device that can minimize the end-to-end execution latency.

Runtime communication. During the runtime, each device needs to fetch the padding data from its neighboring device. Due to the limited computing capability, a device may be still working on generating the output feature map partition when a padding pulling request arrives. To accommodate this case, we block the pulling request until the needed data is prepared. Note that such circumstance is rare since our

TABLE III
DESKTOP PC SPECIFICATIONS

Hardware	Specifications	
CPU	3.60GHz 8-Core Intel i7-7700	
Memory	2666MHz 16GB DDR4	
GPU	GeForce GTX 1050 (Pascal) 640 CUDA core	
Power	Idle	80W
	CPU Fully Loaded	180W
	GPU Fully Loaded	200W

TABLE IV
INFERENCE LATENCY (MS) AND COMPUTATION INTENSITY (CYCLES/KB) OF BASIC IMPLEMENTATION

Model	Raspberry Pi		Jetson TX2		Desktop PC	
	Lat.	Inten.	Lat.	Inten.	Lat.	Inten.
AlexNet	302	615	89	301	46	282
VGG-f	276	563	83	283	44	269
GoogLeNet	769	1568	227	772	114	698
MobileNet	226	461	71	239	37	226

workload partitioning algorithm has optimized the workload allocation to match devices' computing capabilities. Under this plan, the execution time on each device is reasonably close and the utilization of computing resources are maximized as much as possible. Moreover, our workload partitioning algorithm supposes that participated devices can well communicate with each other during the runtime. However, the devices can accidentally break down or temporarily unavailable in real-world deployment. This raises robustness issues, which be discussed in Section VIII.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the performance of CoEdge prototype in terms of inference latency and dynamic energy. We also explore the impact of deadline setting, the system scalability and the adaptability to network fluctuation.

A. Experimental Setup

Prototype. we implement CoEdge prototype with six devices: four Raspberry Pi 3, one Jetson TX2, and one desktop PC, as shown in Fig. 9. The Raspberry Pi 3 and the Jetson TX2 represent weak IoT devices and mobile AI platforms. Besides, we take a desktop PC to emulate small edge servers. The specifications of the three types of devices are provided in Table I, II and III. We employ the Monsoon High Voltage Power Monitor (HVPM) [17] to measure the energy. For bandwidth control, We use the traffic control tool tc [22], which is able to limit the bandwidth under the setting value.

Workload. In our prototype, we use TensorFlow Lite [20] to implement four typical CNN models: AlexNet [19], VGG-f [9], GoogLeNet [38], and MobileNet [39], all of which are trained before deployment. Table IV presents the reported latency of basic implementation and the computing intensity on different platforms. We set the workload as the image classification task on one ImageNet [21] image. The average inference latency and computation intensity of one hundred runs are taken as the results. During the runtime we turn off all applications except for necessary OS background services.

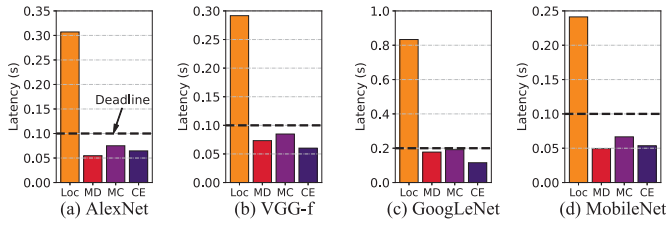


Fig. 10. The end-to-end latency of different approaches running four DNN models. The deadline of AlexNet, VGG-f, GoogLeNet, and MobileNet are set as 100ms, 100ms, 200ms, and 100ms, respectively.

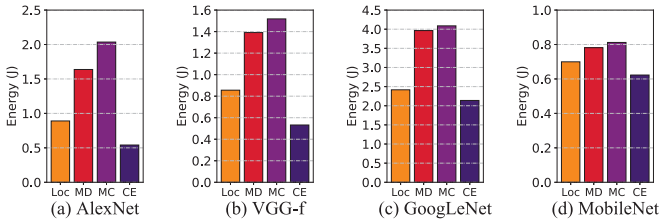


Fig. 11. The dynamic energy consumption of different approaches running four DNN models. We use the testbed in Fig. 9 that consists of four Raspberry Pi 3, one Jetson TX2, and one desktop PC. All experimental settings are the same as that in the experiment of Fig. 10.

Approaches. We compare CoEdge with the following relative approaches. (1) *MoDNN* [40] adopts the same piece-wise partitioning mechanism as CoEdge, but decides partition sizes in proportion to the devices' computing capabilities without considering network conditions. (2) *Musical Chair* [18] is a cooperative inference system that exploits both data and model parallelism. For each layer, it chooses one of the parallelisms and accordingly partitions the workloads in equal proportion. (3) *Local* approach executes CNN inference at the master device solely. In our experiment, the local approach is the baseline, and we fix the master device as a certain Raspberry Pi 3.

B. Performance Comparison

Fig. 10 and Fig. 11 show the latency and dynamic energy results of different models with the local approach (Loc), MoDNN (MD), Musical Chair (MC), and CoEdge (CE). The results in these two figures are measured at the same experimental settings, and the maximum bandwidth between devices are fixed at 1MB/s. We set the deadline for executing the four models as 100ms, 100ms, 200ms, and 100ms, respectively, marked as dashed lines in Fig. 10.

As shown in Fig. 10, CoEdge, Musical Chair and MoDNN always accomplish inference within the deadline. As the most time-consuming option, the local approach is the only one that violates the latency requirement, and CoEdge achieves $7.21 \times \sim 4.49 \times$ latency speedup over it. Comparing the local approach with the other ones reflects the power of cooperative inference gained by harvesting vicinal edge resources. Among the three cooperative approaches, Musical Chair takes higher latency than the other two. This is because that Musical Chair directly split the workload in equal proportion ignoring the resources heterogeneity. CoEdge and MoDNN perform closely

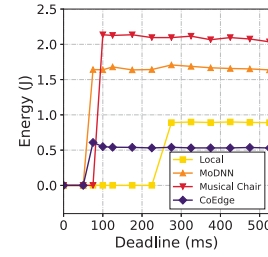


Fig. 12. The dynamic energy consumption of four approaches under varying deadlines built on the testbed in Fig. 9. The result is recorded as zero if the approach fails to finish inference within the deadline.

in the latency metric, but differs their energy costs in the energy metrics.

As an evidence, Fig. 11 shows the least dynamic energy consumption that CoEdge takes comparing with other approaches. CoEdge saves up to 66.9%, 64.9%, 46.0%, and 25.5% energy for four models, respectively (comparing with Musical Chair). To the baseline (local approach), CoEdge saves 39.2%, 37.8%, 11.5%, and 10.9% energy. CoEdge saves energy prominently for AlexNet and VGG-f, but promote not so much for GoogLeNet and MobileNet. This attributes to the structure of CNN models. GoogLeNet's completed block structure comprises a crowd of layers, which incurs frequent data exchanges in cooperative inference. At the opposite end of the spectrum, MobileNet uses a simplified structure and has been well optimized for local inference in embedded devices, which limits the improvement space for cooperative inference. It is worth noting that the local approach consumes less energy than MoDNN and Musical Chair. The reasons come from two aspects. On the one hand, the local approach does not incur communication costs, while MoDNN and Musical Chair need frequent cross-device communication during the runtime, which takes energy. On the other hand, the optimization of MoDNN and Musical Chair does not consider the power characteristics of different types of devices so that the workload are processed in an energy-lavish manner. In contrast, by jointly optimizing the computation-communication tradeoff provided devices' computing capabilities and network conditions, CoEdge achieves the lowest energy costs.

C. Performance Under Varying Deadlines

In this experiment, we explore how the deadline setting impacts the system performance. We run AlexNet to process one image input. The bandwidths between devices are fixed as 1MB/s. Fig. 12 shows the dynamic energy results as a function of deadlines. To emphasize the deadline constraint, we plot the energy result as zero if it fails to accomplish the inference within the deadline. When the latency requirement is very stringent (≤ 50 ms), all approaches miss the deadline. At the 75ms deadline, CoEdge and MoDNN first succeed completing the execution, while CoEdge takes lower energy costs than MoDNN. When the deadline sets 100ms, Musical Chair finishes a full inference, but it consumes higher energy than MoDNN and CoEdge. As the latency requirement gradually relaxes, the local approach finally achieves.

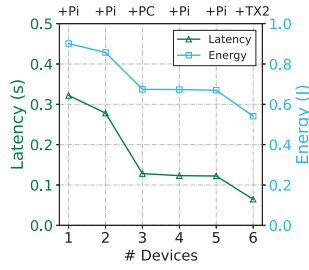


Fig. 13. The latency and dynamic energy results of CoEdge with varying number of devices. The top text indicates which type of device are newly added to the cluster.

Note that CoEdge shows a converged declining curve as the deadline postpones. CoEdge takes higher energy under a stringent deadline (75ms) than a loose deadline (500ms). This is because CoEdge’s optimization puts latency constraint in prior to energy optimization. In the case with a very strict latency requirement, CoEdge prefers to sacrifice some energy-saving to latency reduction. With the requirement loosens, the pressure of satisfying deadline constraint gradually relaxes and CoEdge will transfer emphasis on energy optimization. When the deadline is adequately slack, it is not a constraint to our optimization anymore, in which case the change of that will not impact the workload allocation plan of CoEdge and thus the dynamic energy result keeps stable.

D. Scalability

To evaluate CoEdge’s scalability, we measure the latency and energy by incrementally adding devices to the experimental cluster. We fix the bandwidth as 1MB/s and set a loose deadline of 500ms. The inference task is run AlexNet with one image for classification. We add devices in the following order: Raspberry Pi, Raspberry Pi, desktop PC, Raspberry Pi, Raspberry Pi, and Jetson TX2. Fig. 13 presents the measuring results of CoEdge, where the top text shows the adding devices orderly. With the increase of the cluster scale, both the latency and dynamic energy drop. In particular, there is a distinctive decrease when adding PC (2 → 3) and Jetson TX2 (5 → 6). This is reasonable as the cluster adds a relatively much more powerful device (PC or Jetson) comparing with the Pi at these two points. When the scale is extended to 4 or 5, the latency and dynamic energy keeps approximately stable, indicating that CoEdge runs almost invariant solutions. This is because CoEdge prefers allocating a major portion of the workload to the devices with higher computing capabilities, e.g., the PC. When the cluster already owns such powerful device, adding a weak one (e.g., Pi) will not make distinct changes to the workload allocation plan, and thus the system performance keeps steady as the previous organization.

E. Adaptability to Network Fluctuation

In this experiment, we record the system performance of different cooperative approaches with varying bandwidths. We run AlexNet with one image on the six-device cluster, and the deadline is 100ms, plotted as the dashed line in Fig. 14. For each epoch, CoEdge captures the available bandwidths and triggers a reprogramming on workload partitioning if the bandwidths change. This reprogramming process takes

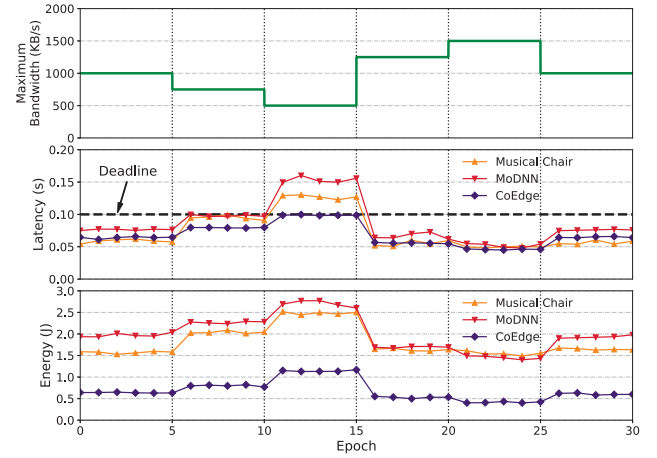


Fig. 14. The latency and dynamic energy results under varying network status. The deadline for CoEdge is set as 100ms.

tiny overhead, reporting less than 10ms in the experiment. We adjust the bandwidth settings between devices in different periods. The top subfigure in Fig. 14 presents the network fluctuation with the bandwidths of 1000KB/s, 750KB/s, 500KB/s, 1250KB/s, 1500KB/s, and 1000KB/s, respectively. As the bandwidth changes, all three approaches vary their performance. The performance variation comes from two reasons. On the one hand, the communication overhead for necessary data exchange during cooperative inference depends on the network conditions. On the other hand, for CoEdge, diverse bandwidths yield diverse partitioning plans and therefore impact the performance of cooperation. In most cases, the latency results of the approaches are approximate. On the energy side, however, CoEdge outperforms other approaches all the time. In particular, when the bandwidth drops at 500KB/s (Epoch 11-15), only CoEdge’s execution satisfies the deadline. In other epochs, as the bandwidth becomes higher, CoEdge adjusts the workload partitioning and the energy costs decrease.

VII. RELATED WORK

Previous research efforts in enabling artificial intelligence at the network edge can be divided into three directions: cloud-assisted execution, local resource exploitation, and multi-device collaboration.

Cloud-assisted execution. Cloud-assisted approaches offload DNN inference workload from local to the cloud fully or partially [10], [41]–[45]. MCDNN [41] fully offloads DNN computation. It creates DNN model variants and selects one from them to maximize the accuracy under resource constraints, while CoEdge does not involve accuracy issues as the model and data are never modified. Neurosurgeon [42] proposes a partially offloading solution, which decides an intermediate partition point in the DNN structure to keep front layers locally and offload rear layers to the cloud. DDNN [44] leverages a similar principle and partitions DNN layers in a cloud-edge-device hierarchy. However, it requests to retrain the DNN model in scheduling, while CoEdge does not require any retraining work. The cloud-assisted approaches have been widely-adopted in mobile scenarios, e.g., drone-enabled vehicles tracking [46],

robotics-based vision applications [47]–[49]. On these specific cases, the cloud’s functionality is further optimized to adjust demands. For example, RILaaS [49] introduces a Robot-Inference-and-Learning-as-a-Service platform with robotics-oriented features such as reliable network protocol, secure authentication, and REST front-end API.

Local resource exploitation. Local approaches keep all computation locally and optimize the performance through hardware specialization or model modification [27], [50]–[53]. Hardware specialization generally centers around basic DNN operations (e.g., matrix multiplication, convolution) to develop efficient hardware accelerators, e.g., ARM ML Processor [54], Google Edge TPU [55]. Some other works target to optimize the utilization of existing hardware. For example, μ Layer [50] accelerates inference in layer granularity by simultaneously utilizing heterogeneous processors inside an edge device. Model modification typically uses model compression technique, e.g., model sparsification and quantization [51]. ReForm [52] reconfigures CNN models by model pruning and selective computing to reduce inference latency on mobile devices. On the same goal, libnumber [53] employs quantization technique to optimize number representation in low-level, reducing both model size and inference latency. CoEdge is orthogonal to such optimizations since it does not apply any structural modifications to the employed DNN.

Multi-device collaboration. Multi-device approaches execute DNN inference using a cluster of devices in the edge environment [14], [40], [56]. Within this category, previous works optimize workload distribution in two ways, layer fusion and workload size adjustment. Layer fusion partitions the feature maps in a fixed pattern, and distributes workload with redundant data - the padding data - to avoid data requests between devices during the runtime. Under this mechanism, DeepThings [14] fuses front convolutional layers and parallelizes these layers on multiple devices. The follow-up work [25] generalizes the fusion operation to all layers and takes resources heterogeneity into account. It designs a dynamic-programming-based fusion searching strategy to adaptively decide which layers are fused and which layers are directly parallelized. Workload size adjustment accommodates the workload allocation to minimize the end-to-end inference latency. MoDNN [40] segments workload greedily and assigns more workload to the devices with higher computing capability without considering the network conditions. Musical Chair [18] introduces a partitioning algorithm integrating data and model parallelism, and partitions the feature maps in equal proportion. Based on Musical Chair, the subsequent work [57]–[59] improves distributed CNN execution in terms of latency performance, scalability, and robustness, respectively.

Our work falls into the multi-device collaboration category, and combines the parallel workflow of layer-fusion techniques [14], [25] and the partitioning mechanism of workload adjustment approaches [18], [40]. Beyond combining the novel designs of these two lines, CoEdge jointly considers available computation and communication resources and improves the workload allocation on heterogeneous devices via an adaptive algorithm, which has not been addressed in prior works.

VIII. DISCUSSION AND FUTURE DIRECTIONS

In this section, we discuss the limitation and extension of CoEdge, and provide some future research directions.

Robustness and Generalization. As a distributed system, crash of any participant or network timeout can result in a system breakdown of cooperative inference. To increase the robustness against such faults, it may be helpful to design modularity [48] for the system or reserve intermediate result backups periodically. Another direction is to further generalize and optimize the system workflow for more sophisticated model structures. Only applying workload partitioning over the whole network may not well fit more complicated architectures as the feature maps of the deeper layers usually exhibit in a smaller height and width.

Other optimizing objectives. CoEdge focuses on optimizing the dynamic energy consumption with preset deadlines for CNN inference. Modifying the objective function of the constrained programming can steer CoEdge to meet other priorities. For example, one can adopt a performance preference by setting a tunable-weighted synthesis of latency and energy. Alternatively, taking static energy consumption into account may produce a more energy-friendly workload allocation plan. Another potential objective is accuracy. Although CoEdge does not sacrifice any accuracy theoretically, running DNNs on some minitype edge devices may still loss precision owing to the limitations of their modest computing capability and the execution mechanism of DNN frameworks. Characterizing and optimizing such accuracy issue is practically significant for edge deployment.

Utilizing edge-oriented resources. Recent technical progresses on edge computing enhancement in computation (e.g., pluggable Google Edge TPU [55], Intel Movidius Neural Compute Stick [60]) could potentially benefit CoEdge performance. For example, by equipping a Raspberry Pi with an Edge TPU, CoEdge may choose to remain the input workload mainly or even completely in situ. This requires more efforts on shaping and utilizing the emerging elastic computing resources. Moreover, improvements on the communication side, e.g., 5G and mmWave, can also boost cooperative edge intelligence.

IX. CONCLUSION

In this paper, we present CoEdge, a distributed DNN computing system that orchestrates cooperative DNN inference over heterogeneous edge devices. We explore the workflow of cooperative inference and formulate it as a constrained optimization problem, which is NP-hard. To solve it efficiently, we design a workload partitioning algorithm to decide efficient partitioning policy in real-time. By jointly optimizing computation and communication, CoEdge can find the optimal workload partitioning plan that minimizes the system energy cost while promising execution latency requirements. Experimental evaluations using a realistic prototype show $7.21\times \sim 4.49\times$ latency speedup over the local approach and up to 25.5% \sim 66.9% energy saving comparing with existing approaches for four widely-adopted DNN models.

REFERENCES

- [1] B. L. R. Stojkoska and K. V. Trivodaliev, "A review of Internet of Things for smart home: Challenges and solutions," *J. Cleaner Prod.*, vol. 140, pp. 1454–1464, Jan. 2017.
- [2] F. Shrouf, J. Ordieres, and G. Miragliotta, "Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm," in *Proc. IEEE Int. Conf. Ind. Eng. Eng. Manage.*, Dec. 2014, pp. 697–701.
- [3] M. Gerla, E.-K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *Proc. IEEE World Forum Internet Things (WF-IoT)*, Mar. 2014, pp. 241–246.
- [4] A.-M. Rahmani *et al.*, "Smart e-Health gateway: Bringing intelligence to Internet-of-Things based ubiquitous healthcare systems," in *Proc. 12th Annu. IEEE Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2015, pp. 826–834.
- [5] Q. Shi and X. Chen, "Carpool for big data: Enabling efficient crowd cooperation in data market for pervasive AI," *IEEE Trans. Veh. Technol.*, vol. 69, no. 7, pp. 7778–7789, Jul. 2020.
- [6] U. R. Acharya *et al.*, "A deep convolutional neural network model to classify heartbeats," *Comput. Biol. Med.*, vol. 89, pp. 389–396, Oct. 2017.
- [7] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
- [8] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 8599–8603.
- [9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [10] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.
- [11] T. Ouyang, Z. Zhou, and X. Chen, "Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 10, pp. 2333–2345, Oct. 2018.
- [12] X. Chen, Q. Shi, L. Yang, and J. Xu, "ThriftyEdge: Resource-efficient edge computing for intelligent IoT applications," *IEEE Netw.*, vol. 32, no. 1, pp. 61–65, Jan. 2018.
- [13] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.
- [14] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.
- [15] RP Foundation. *Raspberry Pi 3*. Accessed: Dec. 15, 2019. [Online]. Available: <https://www.raspberrypi.org/products/raspberry-pi3-model-b/>
- [16] NVIDIA. *NVIDIA Jetson TX*. Accessed: Dec. 15, 2019. [Online]. Available: <https://developer.nvidia.com/embedded/jetson-tx2>
- [17] Monsoon. *High Voltage Power Monitor*. Accessed: May 25, 2019. [Online]. Available: <https://www.monsoon.com/high-voltage-power-monitor>
- [18] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 3709–3716, Oct. 2018.
- [19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
- [20] TensorFlow Benchmark Tool. Accessed: May 15, 2019. [Online]. Available: <https://github.com/tensorflow/tensorflow/tree/r1.4/tensorflow/tools/benchmark>
- [21] 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.*, Jun. 2009, pp. 248–255.
- [22] TL Foundation. *TC—Show/Manipulate Traffic Control Settings*. Accessed: Dec. 15, 2019. [Online]. Available: <https://www.linux.com/tutorials/tc-show-manipulate-traffic-control-settings/>
- [23] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proc. Int. Conf. Learn. Represent.*, 2014, pp. 1–10.
- [24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
- [25] 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.*, Nov. 2019, pp. 195–208.
- [26] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proc. HotCloud*, 2010, vol. 10, no. 4, p. 19.
- [27] S. Yao *et al.*, "FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proc. 16th ACM Conf. Embedded Netw. Sensor Syst.*, Nov. 2018, pp. 278–291.
- [28] Y. Wen, W. Zhang, and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 2716–2720.
- [29] Y. Cui *et al.*, "Software defined cooperative offloading for mobile cloudlets," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1746–1760, Jun. 2017.
- [30] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco, "Distributed inference acceleration with adaptive DNN partitioning and offloading," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Jul. 2020, pp. 854–863.
- [31] M. Mukherjee, V. Kumar, A. Lat, M. Guo, R. Matam, and Y. Lv, "Distributed deep learning-based task offloading for UAV-enabled mobile edge computing," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Jul. 2020, pp. 1208–1212.
- [32] S. Xu *et al.*, "RJCC: Reinforcement-learning-based joint communicational-and-computational resource allocation mechanism for smart city IoT," *IEEE Internet Things J.*, vol. 7, no. 9, pp. 8059–8076, Sep. 2020.
- [33] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Proc. 39th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2012, pp. 37–48.
- [34] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
- [35] G. B. Dantzig, *Linear Programming and Extensions*, vol. 48. Princeton, NJ, USA: Princeton Univ. Press, 1998.
- [36] *V12. 1: User's Manual for CPLEX*, IBM ILOG CPLEX, Int. Bus. Mach. Corp., Armonk, NY, USA, 2009, p. 157, vol. 46, no. 53.
- [37] Google. *gRPC—A RPC Library and Framework*. Accessed: Dec. 15, 2019. [Online]. Available: <https://grpc.io>
- [38] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
- [39] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: <http://arxiv.org/abs/1704.04861>
- [40] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1396–1401.
- [41] 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. 14th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2016, pp. 123–136.
- [42] 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.
- [43] L. Zeng, E. Li, Z. Zhou, and X. Chen, "Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial Internet of Things," *IEEE Netw.*, vol. 33, no. 5, pp. 96–103, Sep. 2019.
- [44] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 328–339.
- [45] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2018, pp. 401–411.
- [46] L. Ballotta, L. Schenato, and L. Carlone, "Computation-communication trade-offs and sensor selection in real-time estimation for processing networks," *IEEE Trans. Netw. Sci. Eng.*, early access, Jul. 9, 2020, doi: [10.1109/TNSE.2020.3008337](https://doi.org/10.1109/TNSE.2020.3008337).
- [47] S. P. Chinchali, E. Cidon, E. Pergament, T. Chu, and S. Katti, "Neural networks meet physical networks: Distributed inference between edge devices and the cloud," in *Proc. 17th ACM Workshop Hot Topics Netw.*, Nov. 2018, pp. 50–56.
- [48] S. Chinchali *et al.*, "Network offloading policies for cloud robotics: A learning-based approach," in *Proc. Robot., Sci. Syst.*, 2019, pp. 1–10.

- [49] A. K. Tanwani, R. Anand, J. E. Gonzalez, and K. Goldberg, "RILaaS: Robot inference and learning as a service," *IEEE Robot. Autom. Lett.*, vol. 5, no. 3, pp. 4423–4430, Jul. 2020.
- [50] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–15.
- [51] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*. [Online]. Available: <http://arxiv.org/abs/1510.00149>
- [52] Z. Xu, F. Yu, C. Liu, and X. Chen, "ReForm: Static and dynamic resource-aware DNN reconfiguration framework for mobile device," in *Proc. 56th Annu. Design Automat. Conf.*, Jun. 2019, pp. 1–6.
- [53] Y. H. Oh *et al.*, "A portable, automatic data quantizer for deep neural networks," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, Nov. 2018, pp. 1–14.
- [54] ARM. *ARM ML Processor*. Accessed: Jul. 16, 2020. [Online]. Available: <https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-n77>
- [55] Google. *Google Edge TPU*. Accessed: Jul. 16, 2020. [Online]. Available: <https://cloud.google.com/edge-tpu>
- [56] D. Hu and B. Krishnamachari, "Fast and accurate streaming CNN inference via communication compression on the edge," in *Proc. IEEE/ACM 5th Int. Conf. Internet-Things Design Implement. (IoTDI)*, Apr. 2020, pp. 157–163.
- [57] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on Internet-of-Things devices," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 4950–4960, Jun. 2020.
- [58] J. Cao *et al.*, "An edge-centric scalable intelligent framework to collaboratively execute DNN," in *Proc. Demo SysML Conf.*, Palo Alto, CA, USA, 2019, pp. 1–2.
- [59] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Robustly executing DNNs in IoT systems using coded distributed computing," in *Proc. 56th Annu. Design Automat. Conf.*, Jun. 2019, pp. 1–2.
- [60] Intel. *Intel Movidius Neural Compute Stick*. Accessed: Jul. 16, 2020. [Online]. Available: <https://software.intel.com/content/www/us/en/develop/hardware/neural-compute-stick.html>



Zhi Zhou (Member, IEEE) received the B.S., M.E., and Ph.D. degrees from the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012, 2014, and 2017, respectively. He is currently an Associate Professor with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. He has been a Visiting Scholar with the University of Göttingen. His research interests include edge computing, cloud computing, and distributed systems. He was nominated for the 2019 CCF Outstanding Doctoral Dissertation Award, a sole recipient of the 2018 ACM Wuhan & Hubei Computer Society Doctoral Dissertation Award, a recipient of the Best Paper Award of the IEEE UIC 2018.



Lei Yang (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Southeast University, Nanjing, China, in 2005 and 2008, respectively, and the Ph.D. degree from the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ, USA, in 2012, where he has been an Assistant Research Professor, since 2013. He was also a Post-Doctoral Scholar with Princeton University, Princeton, NJ, USA. He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of Nevada, Reno, NV, USA. His research interests include stochastic optimization and big data analytics for renewable energy integration, grid integration of plugin electric vehicle, networked control of cyber-physical systems, modeling and control of power systems, network security and privacy, and network optimization and control. He received the Best Paper Award Runner-up of the IEEE INFOCOM 2014.



Liekang Zeng (Graduate Student Member, IEEE) received the B.S. degree in computer science from the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China, in 2018, where he is currently pursuing the Ph.D. degree. His current research interests include mobile edge computing, collaborative cloud-edge computing, and acceleration of deep learning execution on the IoT clusters.



Xu Chen (Senior Member, IEEE) received the Ph.D. degree in information engineering from The Chinese University of Hong Kong in 2012. He was a Post-Doctoral Research Associate with Arizona State University, Tempe, AZ, USA, from 2012 to 2014, and a Humboldt Scholar Fellow with the Institute of Computer Science, University of Göttingen, Germany, from 2014 to 2016. He is currently a Full Professor with Sun Yat-sen University, Guangzhou, China, and the Vice Director of the National and Local Joint Engineering Laboratory of Digital Home

Interactive Applications. He was a recipient of the Prestigious Humboldt Research Fellowship awarded by Alexander von Humboldt Foundation of Germany, the 2014 Hong Kong Young Scientist Runner-Up Award, the 2017 IEEE Communication Society Asia-Pacific Outstanding Young Researcher Award, the 2017 IEEE ComSoc Young Professional Best Paper Award, the Honorable Mention Award of 2010 IEEE international conference on Intelligence and Security Informatics, the Best Paper Runner-Up Award of the 2014 IEEE International Conference on Computer Communications (INFOCOM), and the Best Paper Award of 2017 IEEE International Conference on Communications. He is currently an Area Editor of the IEEE OPEN JOURNAL OF THE COMMUNICATIONS SOCIETY, an Associate Editor of the IEEE TRANSACTIONS WIRELESS COMMUNICATIONS, the IEEE INTERNET OF THINGS JOURNAL, and the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS (JSAC) SERIES ON NETWORK SOFTWAREIZATION AND ENABLERS.



Junshan Zhang (Fellow, IEEE) received the Ph.D. degree from the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, in 2000. He joined the School of Electrical, Computer and Energy Engineering, Arizona State University, in August 2000, where he has been the Fulton Chair Professor, since 2015. His research interests fall in the general field of information networks and data science, including communication networks, the Internet of Things (IoT), fog/edge computing, mobile social networks, and smart grid.

His current research interests include fundamental problems in information networks and data science, including edge computing and its applications in IoT and 5G, IoT data privacy/security, information theory, stochastic modeling, and control for the smart grid. He is a recipient of the ONR Young Investigator Award in 2005 and the NSF CAREER Award in 2003. He received the IEEE Wireless Communication Technical Committee Recognition Award in 2016. His papers have won a few awards, including the Best Student Paper Award at WiOPT 2018, the Kenneth C. Sevcik Outstanding Student Paper Award of ACM SIGMETRICS/IFIP Performance 2016, the Best Paper Runner-Up Award of the IEEE INFOCOM 2009 and IEEE INFOCOM 2014, and the Best Paper Award at the IEEE ICC 2008 and ICC 2017. Building on his research findings, he co-founded Smartiply Inc, a Fog Computing startup company delivering boosted network connectivity and embedded artificial intelligence. He was the TPC Co-Chair for a number of major conferences in communication networks, including the IEEE INFOCOM 2012 and ACM MOBIHOC 2015. He was the General Chair for the ACM/IEEE SEC 2017, WiOPT 2016, and the IEEE Communication Theory Workshop 2007. He was a Distinguished Lecturer of the IEEE Communications Society. He is currently an Editor-in-Chief of the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS and a Senior Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING.