Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing

En Li, Liekang Zeng, Zhi Zhou, *Member, IEEE*, and Xu Chen, *Member, IEEE*

***Abstract*— As a key technology of enabling Artificial Intelligence (AI) applications in 5G era, Deep Neural Networks (DNNs) have quickly attracted widespread attention. However, it is challenging to run computation-intensive DNN-based tasks on mobile devices due to the limited computation resources. What's worse, traditional cloud-assisted DNN inference is heavily hindered by the significant wide-area network latency, leading to poor real-time performance as well as low quality of user experience. To address these challenges, in this paper, we propose Edgent, a framework that leverages edge computing for DNN collaborative inference through device-edge synergy. Edgent exploits two design knobs: (1) DNN partitioning that adaptively partitions computation between device and edge for purpose of coordinating the powerful cloud resource and the proximal edge resource for real-time DNN inference; (2) DNN right-sizing that further reduces computing latency via early exiting inference at an appropriate intermediate DNN layer. In addition, considering the potential network fluctuation in real-world deployment, Edgent is properly design to specialize for both static and dynamic network environment. Specifically, in a static environment where the bandwidth changes slowly, Edgent derives the best configurations with the assist of regression-based prediction models, while in a dynamic environment where the bandwidth varies dramatically, Edgent generates the best execution plan through the online change point detection algorithm that maps the current bandwidth state to the optimal configuration. We implement Edgent prototype based on the Raspberry Pi and the desktop PC and the extensive experimental evaluations demonstrate Edgent's effectiveness in enabling on-demand low-latency edge intelligence.**

***Index Terms*— Edge intelligence, edge computing, deep learning, computation offloading.**

Manuscript received March 26, 2019; revised July 19, 2019; accepted September 25, 2019. Date of publication October 18, 2019; date of current version January 8, 2020. This work was supported in part by the National Science Foundation of China under Grant U1711265, Grant 61972432, and Grant 61802449, 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, in part by the Guangdong Natural Science Funds under Grant 2018A030313032, and in part by the Fundamental Research Funds for the Central Universities under Grant 17lgjc40. This article was presented in part at the 2018 Workshop on Mobile Edge Communications, ACM SIGCOMM MECOMM [1]. The associate editor coordinating the review of this article and approving it for publication was D. Li. *(En Li and Liekang Zeng contributed equally to this work.) (Corresponding author: Xu Chen.)*

The authors are with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China (e-mail: lien5@mail2.sysu.edu.cn; zenglk3@mail2.sysu.edu.cn; zhouzhi9; chenxu35@mail.sysu.edu.cn).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TWC.2019.2946140

I. Introduction

As THE backbone technology supporting modern intelligent mobile applications, Deep Neural Networks (DNNs) represent the most commonly adopted machine learning technique and have become increasingly popular. Benefited by the superior performance in feature extraction, DNN have witnessed widespread success in domains from computer vision [2], speech recognition [3] to natural language processing [4] and big data analysis [5]. Unfortunately, today's mobile devices generally fail to well support these DNN-based applications due to the tremendous amount of computation required by DNN-based applications.

In response to the excessive resource demand of DNNs, traditional wisdom resorts to the powerful cloud datacenter for intensive DNN computation. In this case, the input data generated from mobile devices are sent to the remote cloud datacenter, and the devices receive the execution result as the computation finishes. However, with such cloud-centric approaches, a large amount of data (e.g., images and videos) will be transferred between the end devices and the remote cloud datacenter backwards and forwards via a long wide-area network, which may potentially result in intolerable latency and extravagant energy. To alleviate this problem, we exploit the emerging edge computing paradigm. The principal idea of edge computing [6]–[8] is to sink the cloud computing capability from the network core to the network edges (e.g., base stations and WLAN) in close proximity to end devices [9]–[14]. This novel feature enables computation-intensive and latency-critical DNN-based applications to be executed in a real-time responsive manner (i.e., edge intelligence) [15]. By leveraging edge computing, we can design an on-demand low-latency DNNs inference framework for supporting real-time edge AI applications.

While recognizing the benefits of edge intelligence, our empirical study reveals that the performance of edge-based DNN inference is still highly sensitive to the available bandwidth between the edge server and the mobile device. Specifically, as the bandwidth drops from 1Mbps to 50kbps, the edge-based DNN inference latency increases from 0.123s to 2.317s (detailed in Sec. III-B). Considering the vulnerable and volatile network environment in practical deployment, a natural question is whether we can further optimize the DNN inference under the versatile network environment, especially for some mission-critical DNN-based applications such as intelligent security and industrial robotics [16].

On this issue, in this paper, we exploit the edge computing paradigm and propose Edgent, a low-latency co-inference framework via device-edge synergy. Towards low-latency edge intelligence,[1] Edgent pursues two design knobs. The first is DNN partitioning, which adaptively partitions DNN computation between mobile devices and the edge server according to the available bandwidth so as to utilize the computation capability of the edge server. However, it is insufficient to meet the stringent responsiveness requirement of some mission-critical applications since the execution performance is still restrained by the rest of the model running on mobile devices. Therefore, Edgent further integrate the second knob, DNN right-sizing, which accelerates DNN inference by early exiting inference at an intermediate DNN layer. Essentially, the early-exit mechanism involves a latency-accuracy tradeoff. To strike a balance on the tradeoff with the existing resources, Edgent makes a joint optimization on both DNN partitioning and DNN right-sizing in an on-demand manner. Specifically, for mission-critical applications that are typically with a pre-defined latency requirement, Edgent maximizes the accuracy while promising the latency requirement.

Considering the versatile network condition in practical deployment, Edgent further develops a tailored configuration mechanism so that Edgent can pursue better performance in both static network environment and dynamic network environment. Specifically, in a static network environment (e.g., local area network with fiber or mmWave connection), we regard the bandwidth as stable and figure out a collaboration strategy through execution latency estimation based on the current bandwidth. In this case, Edgent trains regression models to predict the layer-wise inference latency and accordingly derives the optimal configurations for DNN partitioning and DNN right-sizing. In a dynamic network environment (e.g., 5G cellular network, vehicular network), to alleviate the impact of network fluctuation, we build a look-up table by profiling and recording the optimal selection of each bandwidth state and specialize the runtime optimizer to detect the bandwidth state transition and map the optimal selection accordingly. Through our specialized design for different network environment, Edgent is able to maximize the inference accuracy without violating the application responsiveness requirement. The prototype implementation and extensive evaluations based on the Raspberry Pi and the desktop PC demonstrate Edgent's effectiveness in enabling on-demand low-latency edge intelligence.

To summarize, we present the contribution of this paper as follows:

- We propose Edgent, a framework for on-demand DNN collaborative inference through device-edge synergy, in which we jointly optimize DNN partitioning and DNN right-sizing to maximize the inference accuracy while promising application latency requirement.
- Considering the versatile network environments (i.e., static network environment and dynamic network environment), we specialize the workflow design for Edgent to achieve better performance.
- We implement and experiment Edgent prototype with a Raspberry Pi and a desktop PC. The evaluation results based on the real-world network trace dataset demonstrate the effectiveness of the proposed Edgent framework.

The rest of this paper is organized as follows. First, we review the related literature in Sec. II and present the background and motivation in Sec. III. Then we propose the design of Edgent in Sec. IV. The results of the performance evaluation are shown in Sec. V to demonstrate the effectiveness of Edgent. Finally, we conclude in Sec. VI.

II. Related Work

Discussions on the topic of mobile DNN computation have recently obtained growing attention. By hosting artificial intelligence on mobile devices, mobile DNN computation deploys DNN models close to users in order to achieve more flexible execution as well as more secure interaction [15]. However, it is challenging to directly execute the computation-intensive DNNs on mobile devices due to the limited computation resource. On this issue, existing efforts dedicate to optimize the DNN computation on edge devices.

Towards low-latency and energy-efficient mobile DNN computation, there are mainly three ways in the literature: runtime management, model architecture optimization and hardware acceleration. Runtime management is to offload computation from mobile devices to the cloud or the edge server, which utilizes external computation resource to obtain better performance. Model architecture optimization attempts to develop novel DNN structure so as to achieve desirable accuracy with moderate computation [17]–[21]. For example, aiming at reducing resource consumption during DNN computation, DNN models are compressed by model pruning [22]–[24]. Recent advance in such kind of optimization has turned to Network Architecture Search (NAS) [25]–[27]. Hardware acceleration generally embraces basic DNN computation operations in hardware level design [28]–[30] while some works aim at optimizing the utilization of the existing hardware resources [31]–[33].

As one of the runtime optimization methods, DNN partitioning technology segments specific DNN model into some successive parts and deploys each part on multiple participated devices. Specifically, some frameworks [34]–[36] take advantage of DNN partitioning to optimize the computation offloading between the mobile devices and the cloud, while some framework target to distribute computation workload among mobile devices [37]–[39]. Regardless of how many devices are involved, DNN partitioning dedicates to maximize the utilization of external computation resources so as to accelerate the mobile computation. As for DNN right-sizing, it focuses on adjusting the model size under the limitation of the existing environment. On this objective, DNN right-sizing appeals to specialized training technique to generate the deuterogenic branchy DNN model from the original standard DNN model. In this paper, we implement the branchy model with the assist of the open-source BranchyNet [40] framework and Chainer [41] framework.

[1]As an initial study, in this work, we focus on the problem of execution latency optimization. Energy efficiency will be considered in future studies.

Compared with the existing work, the novelty of our framework is summarized in the following three aspects. First of all, given the pre-defined application latency requirement, Edgent maximizes the inference accuracy according to the available computation resources, which is significantly different from the existing studies. This feature is essential for practical deployment since different DNN-based applications may require different execution deadlines under different scenarios. Secondly, Edgent integrates both DNN partitioning and DNN right-sizing to maximize the inference accuracy while promising application execution deadline. It is worth noting that neither the model partitioning nor the model right-sizing can well address the timing requirement challenge solely. For model partitioning, it does reduce the execution latency, whereas the total processing time is still restricted by the part on the mobile device. For model right-sizing, it accelerates the inference processing through the early-exit mechanism, but the total computation workload is still dominated by the original DNN model architecture and thus it is hard to finish the inference before the application deadline.

Therefore, we propose to integrate these two approaches to expand the design space. The integration of model partitioning and model right-sizing is not a one-step effort, and we need to carefully design the decision optimization algorithms to fully explore the selection of the partition point and the exiting point and thus to strike a good balance between accuracy and latency in an on-demanded manner. Through these efforts, we can achieve the design target such that given the predefined latency constraint, the DNN inference accuracy is maximized without violating the latency requirement. Last but not least, we specialize the design of Edgent for both static and dynamic network environments while existing efforts (e.g., [38]) mainly focus on the scenario with stable network. Considering the diverse application scenarios and deployment environments in practice, we specialize the design of the configurator and the runtime optimizer for the static and dynamic network environments, by which Edgent can generate proper decisions on the exit point and partition point tailored to the network conditions.

III. Background and Motivation

In this section, we first give a brief introduction on DNN. Then we analyze the limitation of edge- and device-only methods, motivated by which we explore the way to utilize DNN partitioning and right-sizing to accelerate DNN inference with device-edge synergy.

A. A Brief Introduction on DNN

With the proliferation of data and computation capability, DNN has served as the core technology for a wide range of intelligent applications across Computer Vision (CV) [2] and Natural Language Processing (NLP) [4]. Fig. 1 shows a toy DNN for image recognition that recognizes a cat. As we can see, a typical DNN model can be represented as a directed graph, which consists of a series of connected layers where neurons are connected with each other. During the DNN computation (i.e., DNN training or DNN inference), each neuron accepts weighted inputs from its neighborhood and generates outputs after some activation operations. A typical DNN may have dozens of layers and hundreds of nodes per layer and the total number of parameters can easily reach to the millions level, thus a typical DNN inference demands a large amount of computation. In this paper, we focus on DNN inference rather than DNN training since the training process is generally delay-tolerant and often done offline on powerful cloud datacenters.

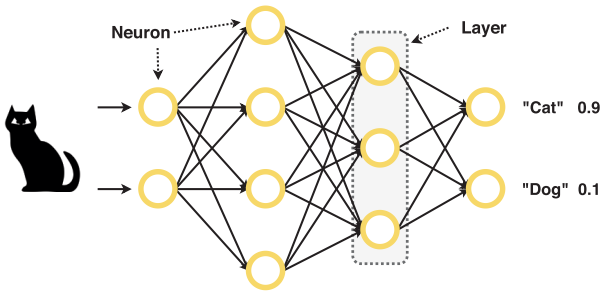


Fig. 1. A four-layer DNN for image classification.

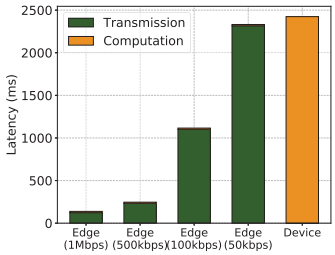


Fig. 2. Execution runtime of edge-only (Edge) and device-only (Device) approaches of AlexNet under different bandwidths.

B. Insufficiency of Device- or Edge-Only DNN Inference

Traditional mobile DNN computation is either solely performed on mobile devices or wholly offloaded to cloud/edge servers. Unfortunately, both approaches may lead to poor performance (i.e., high end-to-end latency), making it difficult to meet real-time applications latency requirements [10]. For illustration, we employ a Raspberry Pi and a desktop PC to emulate the mobile device and edge server respectively, and perform image recognition task over cifar-10 dataset with the classical AlexNet model [42]. Fig. 2 depicts the end-to-end latency subdivision of different methods under different bandwidths on the edge server and the mobile device (simplified as Edge and Device in Fig. 2). As shown in Fig. 2, it takes more than 2s to finish the inference task on a resource-constrained mobile device. As a contrast, the edge server only spends 0.123s for inference under a 1Mbps bandwidth. However, as the bandwidth drops, the execution latency of the edge-only method climbs rapidly (the latency climbs to 2.317s when the bandwidth drops to 50kbps). This indicates that the performance of the edge-only method is dominated by the data transmission latency (the computation time at server side remains at $\sim$10ms) and is therefore highly sensitive to the available bandwidth. Considering the scarcity of network

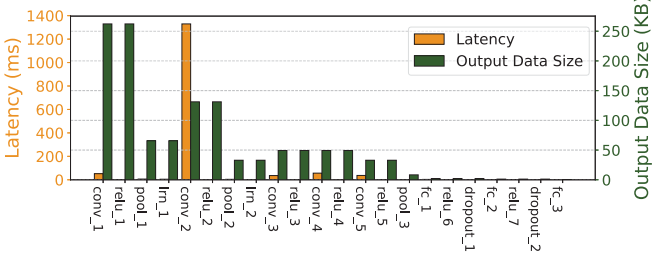


Fig. 3. AlexNet layer-wise runtime and output data size on Raspberry Pi.

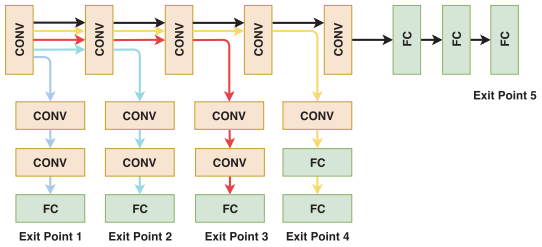


Fig. 4. A branchy AlexNet model with five exit points.

bandwidth resources in practice (e.g., due to network resource contention between users and applications) and the limitation of computation resources on mobile devices, device- and edge-only methods are insufficient to support emerging mobile applications in stringent real-time requirements.

C. DNN Partitioning and Right-Sizing towards Edge Intelligence

DNN Partitioning: To better understand the performance bottleneck of DNN inference, in Fig. 3 we refine the layer-wise execution latency (on Raspberry Pi) and the intermediate output data size per layer. Seen from Fig. 3, the latency and output data size of each layer show great heterogeneity, implying that a layer with a higher latency may not output larger data size. Based on this observation, an intuitive idea is DNN partitioning, i.e., dividing DNN into two parts and offloading the computation-intensive part to the server at a low transmission cost and thus reducing total end-to-end execution latency. As an illustration, we select the second local response normalization layer (i.e., `lrn_2`) in Fig. 3 as the partition point and the layers before the point are offloaded to the server side while the rest remains on the device. Through model partitioning between device and edge, hybrid computation resources in proximity can be in comprehensive utilization towards low-latency DNN inference.

DNN Right-Sizing: Although DNN partitioning can significantly reduce the latency, it should be noted that with the optimal DNN partitioning, the inference latency is still constrained by the remaining computation on the mobile device. To further reduce the execution latency, the DNN right-sizing method is employed in conjunction with DNN partitioning. DNN right-sizing accelerates DNN inference through the early-exit mechanism. For example, by training DNN models with multiple exit points, a standard AlexNet model can be derived as a branchy AlexNet as Fig. 4 shows, where a shorter branch (e.g., the branch ended with the exit point 1) implies a smaller model size and thus a shorter runtime. Note that in Fig. 4 only the convolutional layers (CONV) and the fully-connected layers (FC) are drawn for the ease of illustration. This novel branchy structure demands novel training methods. In this paper, we implement the branchy model training with the assist of the open-source BranchyNet [40] framework.

Problem Definition: Clearly, DNN right-sizing leads to a latency-accuracy tradeoff, i.e., while the early-exit mechanism reduces the total inference latency, it hurts the inference accuracy. Considering the fact that some latency-sensitive applications have strict deadlines but can tolerate moderate precision losses, we can strike a good balance between latency and accuracy in an on-demand manner. In particular, given the pre-defined latency requirement, our framework should maximize accuracy within the deadline. More precisely, the problem addressed in this paper can be summarized as how to make a joint optimization on DNN partitioning and DNN right-sizing in order to maximize the inference accuracy without violating the pre-defined latency requirement.

IV. Framework and Design

In this section, we present the design of Edgent, which generates the optimal collaborative DNN inference plan that maximizes the accuracy while meeting the latency requirement in both the static and dynamic bandwidth environment.

A. Framework Overview

Edgent aims to pursue a better DNN inference performance across a wide range of network conditions. As shown in Fig. 5, Edgent works in three stages: offline configuration stage, online tuning stage and co-inference stage.

At the **offline configuration stage**, Edgent inputs the employed DNN to the *Static/Dynamic Configurator* component and obtains the corresponding configuration for online tuning. To be specific, composed of trained regression models and branchy DNN model, the static configuration will be employed where the bandwidth keeps stable during the DNN inference (which will be detailed in Sec. IV-B), while composed of the trained branchy DNN and the optimal selection for different bandwidth states, the dynamic configuration will be used adaptive to the state dynamics (which will be detailed in Sec. IV-C).

At the **online tuning stage**, Edgent measures the current bandwidth state and makes a joint optimization on DNN partitioning and DNN right-sizing based on the given latency requirement and the configuration obtained offline, aiming at maximizing the inference accuracy under the given latency requirement.

At the **co-inference stage**, based on the co-inference plan (i.e., the selected exit point and partition point) generated at the online tuning stage, the layers before the partition point will be executed on the edge server with the rest remaining on the device.

During DNN inference, the bandwidth between the mobile device and the edge server may be relatively stable or frequently changing. Though Edgent runs on the same workflow

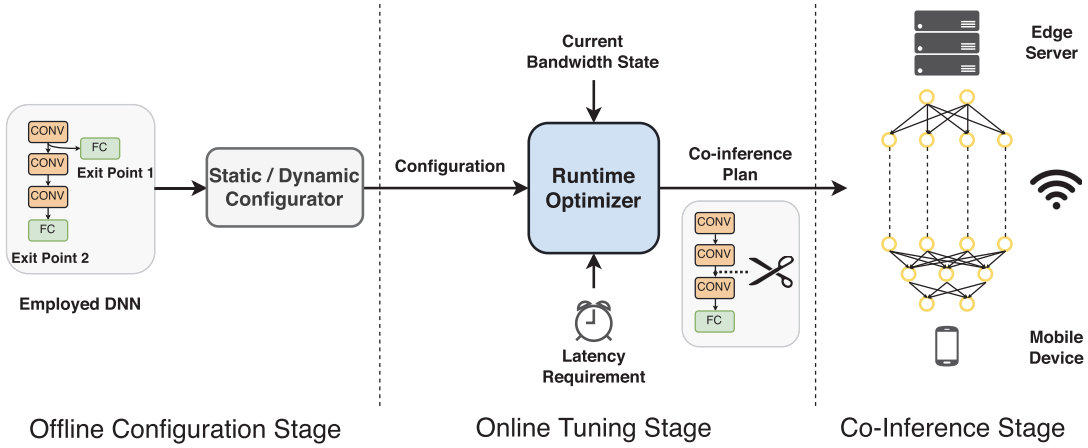


Fig. 5. Edgent framework overview.

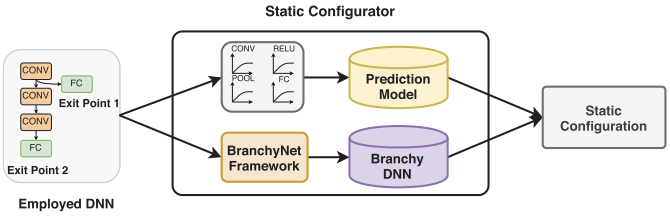


Fig. 6. The static configurator of Edgent.

TABLE I
THE INDEPENDENT VARIABLES OF PREDICTION MODELS

Type of DNN Layer	Independent Variable(s)
Convolutional	# of input feature maps, (filter size/stride)^2×(# of filters)
Relu	input data size
Pooling	input data size, output data size
Local Response Normalization	input data size
Dropout	input data size
Fully-Connected	input data size, output data size

in both static and dynamic network environments, the function of *Configurator* component and *Runtime Optimizer* component differ. Specifically, under a static bandwidth environment, the configurator trains regression models to predict inference latency and the branchy DNN to enable early-exit mechanism. The static configuration generated offline includes the trained regression models and the trained branchy DNN, based on which the *Runtime Optimizer* will figure out the optimal co-inference plan. Under a dynamic bandwidth environment, the dynamic configurator creates a configuration map that records the optimal selection for different bandwidth state via the change point detector, which will then be input to the *Runtime Optimizer* to generate the optimal co-inference plan. In the following, we will discuss the specialized design of the configurator and optimizer for the static and dynamic environments, respectively.

B. Edgent for Static Environment

As a starting point, we first consider our framework design in the case of the static network environment. The key idea of the static configurator is to train regression models to predict the layer-wise inference latency and train the branchy model to enable early-exit mechanism. The configurator specialized for the static bandwidth environment is shown in Fig. 6.

At the **offline configuration stage**, to generate static configuration, the static configurator initiates two tasks: (1) profile layer-wise inference latency on the mobile device and the edge server respectively and accordingly train regression models for different kind of DNN layer (e.g., `Convolution`, `Fully-Connected`, etc.), (2) train the DNN model with multiple exit points via BranchyNet framework to obtain branchy DNN. The profiling process is to record the inference latency of each type of layers rather than that of the entire model. Base on the profiling results, we establish the prediction model for each type of layers by performing a series of regression models with independent variables shown in Table I. Since there are limited types of layers in a typical DNN, the profiling cost is moderate. Since the layer-wise inference latency is infrastructure-dependent and the DNN training is application-related, for a specific DNN inference Edgent only needs to initialize the above two tasks for once.

At the **online tuning stage**, using the static configuration (i.e., the prediction model and the branchy DNN), the *Runtime Optimizer* component searches the optimal exit point and partition point to maximize the accuracy while ensuring the execution deadline with three inputs: (1) the static configuration, (2) the measured bandwidth between the edge server and the end device and (3) the latency requirement. The search process of joint optimization on the selection of partition point and exit point is described in *Algorithm 1*. For a DNN model with M exit points, we denote that the branch of i-th exit point has N_i layers and D_p is the output of the p-th layer. We use the abvoe-mentioned regression models to predict the latency ED_j of the j-th layer running on the device and the latency ES_j of the j-th layer running on the server. Under a certain bandwidth B, fed with the input data $Input$, we can calculate the total latency $A_{i,p}$ by summing up the computation latency on each side and the communication

latency for transferring input data and intermediate execution result. We denote $p\text{-}th$ layer as the partition point of the branch with the $i\text{-}th$ exit point. Therefore, $p = 1$ indicates that the total inference process will only be executed on the device side (i.e., $ES_p = 0, D_{p-1}/B = 0, Input/B = 0$) whereas $p = N_i$ means the total computation is only done on the server side (i.e., $ED_p = 0, D_{p-1}/B = 0$). Through the exhaustive search on points, we can figure out the optimal partition point with the minimum latency of the $i\text{-}th$ exit point. Since the model partitioning does not affect the inference accuracy, we can successively test the DNN inference with different exit layers (i.e., with different precision) and find the model with the maximum accuracy while satisfying the latency requirement at the same time. As the regression models for layer-wise latency prediction have been trained in advance, *Algorithm 1* mainly involves linear search operations and can be completed very quickly (no more than 1ms in our experiment).

There are two basic assumptions for our design. One is that we assume the existing DNN inference on mobile devices cannot satisfy the application latency requirement and there is an edge server in proximity that is available to be employed to accelerate DNN inference through computation offloading. The other assumption is that the regression models for performance prediction are trained based on the situation that the computation resources for the DNN model execution on the mobile device and the edge server are fixed and allocated beforehand. Nevertheless, these assumptions can be further relaxed, since we can train more advanced performance prediction models (e.g., using deep learning models) by taking different resource levels into account.

Algorithm 1 Runtime Optimizer for Static Environment

Input:
M: the number of exit points in the DNN model
$\{N_i | i = 1, \cdots, M\}$: the number of layers in the branch of exit pint i
$\{L_j | j = 1, \cdots, N_i\}$: the layers in the branch of exit point i
$\{D_j | j = 1, \cdots, N_i\}$: layer-wise output data size in the branch of exit point i
$f(L_j)$: the prediction model that returns the $j\text{-}th$ layer's latency
B: current available bandwidth
$Input$: input data size
$Latency$: latency requirement
Output:
Selection of exit point and partition point

Procedure
for $i = M, \cdots, 1$ **do**
  Select the branch of $i\text{-}th$ exit point
  for $j = 1, \cdots, N_i$ **do**
    $ES_j \leftarrow f_{edge}(L_j)$
    $ED_j \leftarrow f_{device}(L_j)$
  end for
  $A_{i,p} = \underset{p=1,\cdots,N_i}{\operatorname{argmin}} (\sum_{j=1}^{p-1} ES_j + \sum_{k=p}^{N_i} ED_j + Input/B + D_{p-1}/B)$
  if $A_{i,p} \leq Latency$ **then**
    return Exit point i and partition point p
  end if
end for
return NULL ▷ can not meet the latency requirement

C. Edgent for Dynamic Environment

The key idea of Edgent for the dynamic environment is to exploit the historical bandwidth traces and employ *Configuration Map Constructor* to generate the optimal co-inference plans for versatile bandwidth states in advance. Specifically, under the dynamic environment Edgnet generates the dynamic configuration (i.e., a configuration map that records the optimal selections for different bandwidth states) at the offline stage, and at the online stage Edgent searches for the optimal partition plan according to the configuration map. The configurator specialized for the dynamic bandwidth environment is shown in Fig. 7.

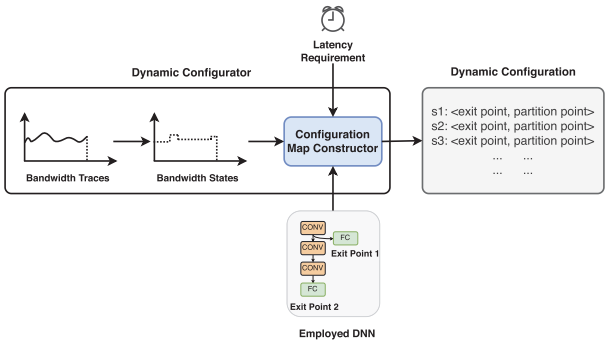


Fig. 7. The dynamic configurator of Edgent.

At the **offline configuration stage**, the dynamic configurator performs following initialization: (1) sketch the bandwidth states (noted as $s1, s2, \cdots$) from the historical bandwidth traces and (2) pass the bandwidth states, latency requirement and the employed DNN to the static Edgent to acquire the optimal exit point and partition point for the current input. The representation of the bandwidth states is motivated by the existing study of adaptive video streaming [43], where the throughput of TCP connection can be modeled as a piece-wise stationary process where the connection consists of multiple non-overlapping stationary segments. In the dynamic configurator, it defines a bandwidth state s as the mean of the throughput on the client side in a segment of the underlying TCP connection. With each bandwidth state, we acquire the optimal co-inference plan by calling the *Configuration Map Constructor* and record that in a map as the dynamic configuration.

The configuration map construction algorithm run in *Configuration Map Constructor* is presented in *Algorithm 2*. The key idea of *Algorithm 2* is to utilize the reward function to evaluate the selection of the exit point and partition point. Since our design goal is to maximize the inference accuracy while promising the application latency requirement, it is necessary to measure whether the searched co-inference strategy can meet the latency requirement and whether the inference accuracy has been maximized. Therefore, we define a reward

Algorithm 2 Configuration Map Construction

Input:
$\{s_i|i = 1, \cdots, N\}$: the bandwidth states
$\{C_j|j = 1, \cdots, M\}$: the co-inference strategy
$R(C_j)$: the reward of co-inference strategy C_j
Output:
Configuration Map

Procedure
for $i = 1, \cdots, N$ **do**
  Select the bandwidth state s_i
  $reward_{max} = 0$, $C_{optimal} = 0$
  for $j = 1, \cdots, M$ **do**
    $reward_{c_j} \leftarrow R(C_j)$
    if $reward_{max} \leq reward_{c_j}$ **then**
      $reward_{max} = reward_{c_j}$, $C_{optimal} = C_j$
    end if
  end for
  Get the corresponding $exit\,point$ and $partition\,point$ of $C_{optimal}$
  Add $S_i :< exit\,point, partition\,point >$ to the Configuration Map
end for
return Configuration Map

to evaluate the performance of each search step as follow:

$$reward_{step} = \begin{cases} \exp(acc) + throughput, & t_{step} \leq t_{req}, \\ 0, & else, \end{cases} \tag{1}$$

where t_{step} is the average execution latency in the current search step (i.e., the selected exit point and partition point in the current search step), which equals to $\frac{1}{throughput}$. The conditions of Equation (1) prioritizes that the latency requirement t_{req} should be satisfied, otherwise the reward will be set as 0 directly. Whenever the latency requirement is met, the reward of the current step will be calculated as $\exp(acc) + throughput$, where acc is the accuracy of current inference. If the latency requirement is satisfied, the search emphasizes on improving the accuracy and when multiple options have the similar accuracy, the one with the higher throughput will be selected. In *Algorithm 2*, s_i represents a bandwidth state extracted from the bandwidth traces and C_j is a co-inference strategy (i.e., a combination of exit point and partition point) indexed by j. $R(C_j)$ denotes the reward of the co-inference strategy C_j, which can be obtained by calculating Equation (1) according to the accuracy and the throughput of C_j.

At the **online tuning stage**, the *Runtime Optimizer* component selects the optimal co-inference plan according to the dynamic configuration and real-time bandwidth measurements. *Algorithm 3* depicts the whole process in *Runtime Optimizer*. Note that *Algorithm 3* calls the change point detection function $D(B_{1,\cdots,t})$ [44] to detect the distributional state change of the underlying bandwidth dynamics. Particularly, when the sampling distribution of the bandwidth measurement has changed significantly, the change point detection function records a change point and logs a bandwidth state transition. Then with $find(state)$ function, the *Runtime Optimizer* captures the corresponding co-inference strategy to the current bandwidth state (or the closest state) in the dynamic configuration and accordingly guides the collaborative inference process at the co-inference stage.

V. Performance Evaluation

In this section, we present our implementation of Edgent and the evaluation results.

A. Experimental Setup

We implement a prototype based on the Raspberry Pi and the desktop PC to demonstrate the feasibility and efficiency of Edgent. Equipped with a quad-core 3.40 GHz Intel processor and 8 GB RAM, a desktop PC is served as the edge server. Equipped with a quad-core 1.2 GHz ARM processor and 1 GB RAM, a Raspberry Pi 3 is used to act as a mobile device.

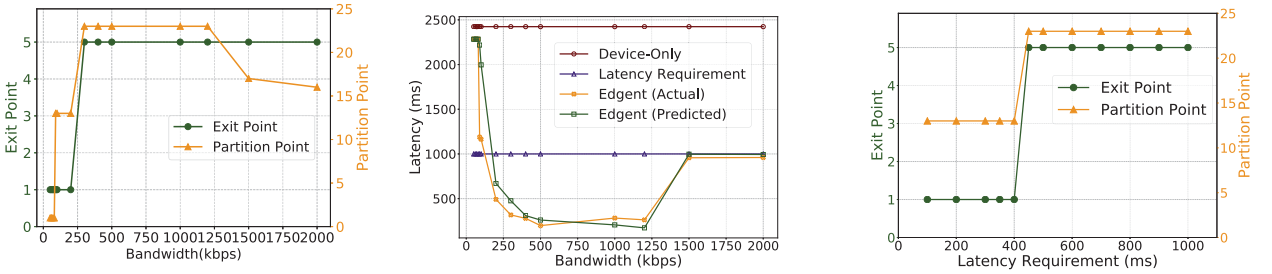
To set up a static bandwidth environment, we use the WonderShaper tool [45] to control the available bandwidth. As for the dynamic bandwidth environment setting, we use the dataset of Belgium 4G/LTE bandwidth logs [46] to emulate the online dynamic bandwidth environment. To generate the configuration map, we use the synthetic bandwidth traces provided by Oboe [43] to generate 428 bandwidth states range from 0Mbps to 6Mbps.

To obtain the branchy DNN, we employ BranchyNet [40] framework and Chainer [41] framework, which can well support multi-branchy DNN training. In our experiments we take the standard AlexNet [42] as the toy model and train the AlexNet model with five exit points for image classification over the cifar-10 dataset [47]. As shown in Fig. 4, the trained branchy AlexNet has five exit points, with each point corresponds to a branch of the standard AlexNet. From the longest branch to the shortest branch, the number of layers in each exit point is 22, 20, 19, 16 and 12, respectively.

B. Experiments in Static Bandwidth Environment

In the static configurator, the prediction model for layer-wise prediction is trained based on the independent variables presented in Table I. The branchy AlexNet is deployed on both the edge server and the mobile device for performance evaluation. Specifically, due to the high-impact characteristics of the latency requirement and the available bandwidth during the optimization procedure, the performance of Edgent is measured under different pre-defined latency requirements and varying available bandwidth settings.

We first explore the impact of the bandwidth by fixing the latency requirement at 1000ms and setting the bandwidth from 50kbps to 1.5Mbps. Fig. 8(a) shows the optimal co-inference plan (i.e., the selection of partition point and exit point) generated by Edgent under various bandwidth settings. Shown in Fig. 8(a), as bandwidth increases, the optimal exit point becomes larger, indicating that a better network environment leads to a longer branch of the employed DNN and thus higher accuracy. Fig. 8(b) shows the inference latency



(a) Selection under different bandwidths (b) Model runtime under different bandwidths (c) Selection under different latency requirements

Fig. 8. The results under different bandwidths and different latency requirements.

Algorithm 3 Runtime Optimizer for Dynamic Environment

Input:

$\{B_{1,\cdots,t}\}$: the accumulated bandwidth measurements until the current moment t

$\{C_j|j=1,\cdots,t\}$: the co-inference strategy

$\{s_i|i=1,\cdots,t\}$: the bandwidth states

$D(B_{1,\cdots,t})$: the bandwidth state detection function that returns the current bandwidth state

$find(s)$: find the co-inference strategy corresponds to the given state s

Output:

Co-inference strategy

Procedure
$C_t = C_{t-1}$
$s_t = D(B_{i,\cdots,t})$
if $s_t \neq s_{t-1}$ **then**
    $C_t \leftarrow find(s_t)$
end if
$s_{t-1} = s_t$
$C_{t-1} = C_t$
return C_t

change trend where the latency first descends sharply and then climbs abruptly as the bandwidth increases. This fluctuation makes sense since the bottleneck of the system changes as the bandwidth becomes higher. When the bandwidth is smaller than 250kbps, the optimization of Edgent is restricted by the poor communication condition and prefers to trade the high inference accuracy for low execution latency, for which the exit point is set as 3 rather than 5. As the bandwidth rises, the execution latency is no longer the bottleneck so that the exit point climbs to 5, implying a model with larger size and thus higher accuracy. There is another interesting result that the curve of predicted latency and the measured latency is nearly overlapping, which shows the effectiveness of our regression-based prediction. Next, we set the available bandwidth at 500kbps and vary the latency requirement from 1000ms to 100ms for further exploration. Fig. 8(c) shows the optimal partition point and exit point under different latency requirements. As illustrated in Fig. 8(c), both the optimal exit point and partition point climb higher as the latency requirement relaxes, which means that a later execution deadline will provide more room for accuracy improvement.

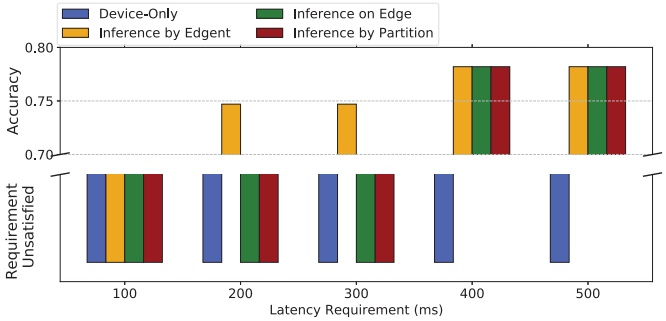


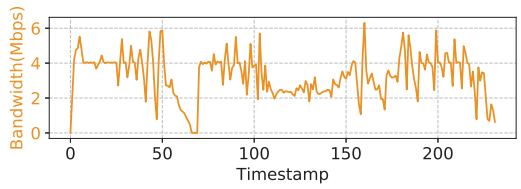
Fig. 9. The accuracy comparison under various latency requirement.

Fig. 9 shows the model inference accuracy of different methods under different latency requirement settings (the bandwidth is fixed as 400kbps). The accuracy is described in negative when the method cannot satisfy the latency requirement. As Fig. 9 shows, given a tightly restrict latency requirement (e.g., 100ms), all the four methods fail to meet the requirement, for which all the four squares lay below the standard line. However, as the latency requirement relaxes, Edgent works earlier than the other three methods (at the requirements of 200ms and 300ms) with the moderate loss of accuracy. When the latency requirement is set longer than 400ms, all the methods except for device-only inference successfully finish execution in time.

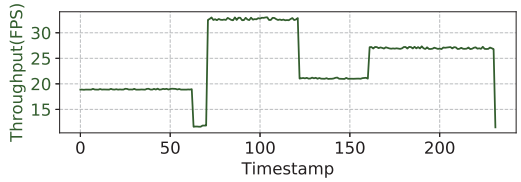
C. Experiments in Dynamic Bandwidth Environment

For the configuration map generation, we use the bandwidth traces provided in Oboe [43]. Each bandwidth trace in the dataset consists of 49 pairs of data tuple about download chunks, including start time, end time and the average bandwidth. We calculate the mean value of all the average bandwidth in the same bandwidth trace to represent the bandwidth state fluctuation, from which we obtain 428 bandwidth states range from 0Mbps to 6Mbps. According to the *Algorithm 2*, through the exhaustive search, we figure out the optimal selection of each bandwidth state. The latency requirement in this experiment is also set to 1000ms.

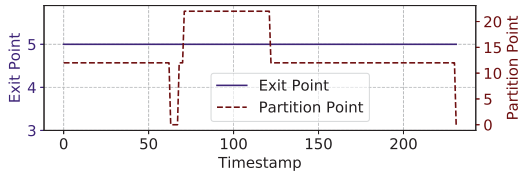
For online change point detection, we use the existing implementation [48] and integrate it with the *Runtime Optimizer*. We use the Belgium 4G/LTE bandwidth logs dataset [46] to perform online bandwidth measurement, which records the bandwidth traces that are measured on several types of



(a) A example bandwidth trace on Belgium 4G/LTE dataset [46]



(b) The throughput of DNN model inference



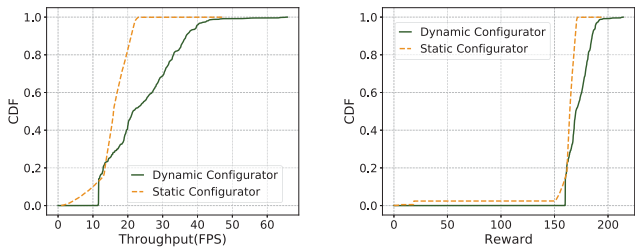
(c) The selection of exit point and partition point at each timestamp in the bandwidth traces

Fig. 10. An example showing the decision-making ability of Edgent in a bandwidth traces recorded on bus.

transportation: on foot, bicycle, bus, train or car. Additionally, Since that most of the bandwidth logs are over 6Mbps and in some cases even up to 95Mbps, to adjust the edge computing scenario, in our experiment, we scale down the bandwidth of the logs and limit it in a range from 0Mbps to 10Mbps.

In this experiment Edgent runs in a dynamic bandwidth environment emulated by the adjusted Belgium 4G/LTE bandwidth logs. Fig. 10(a) shows an example bandwidth trace on the dataset that is recorded on a running bus. Fig. 10(b) shows the DNN model inference throughput results under the bandwidth environment showed in Fig. 10(a). The corresponding optimal selection of the exit point and partition point is presented in Fig. 10(c). Seen from Fig. 10(c), the optimal selection of model inference strategy varies with the bandwidth changes but the selected exit point stays at 5, which means that the network environment is good enough for Edgent to satisfy the latency requirement though the bandwidth fluctuates. In addition, since the exit point remains invariable, the inference accuracy also keeps stable. Dominated by our reward function design, the selection of partition points approximately follows the traces of the throughput result. The experimental results show the effectiveness of Edgent under the dynamic bandwidth environment.

We further compare the static configurator and the dynamic configurator under the dynamic bandwidth environment in Fig. 11. We set the latency requirement as 1000ms and record the throughput and the reward for the two configurators, base on which we calculate the Cumulative Distribution Function (CDF). Seen from Fig. 11(a), under the same CDF level, Edgent with the dynamic configurator achieves higher



(a) Throughput (b) Reward

Fig. 11. The throughput and reward comparison between two configurations.

throughput, demonstrating that under the dynamic network environment the dynamic configurator performs co-inference with higher efficiency. For example, set CDF as 0.6, the dynamic configurator makes 27 FPS throughput while the static configurator makes 17 FPS. In addition, the CDF curve of dynamic configurator rises with 11 FPS throughput while the static configurator begins with 1 FPS, which indicates that the dynamic configurator works more efficiently than the static configurator at the beginning.

Fig. 11(b) presents the CDF results of reward. Similarly, under the same CDF level, the dynamic configurator acquires higher reward than the static configurator and the CDF curve of the dynamic configurator rises latter again. However, in Fig. 11(b) the two curves is closer than in Fig. 11(a), which means that the two configurators achieve nearly the same good performance from the perspective of reward. This is because the two configurators make similar choices in the selection of exit point (i.e., in most cases both of them select exit point 5 as part of the co-inference strategy). Therefore the difference of the reward mainly comes from the throughput result. It demonstrates that the static configurator may perform as well as the dynamic configurator in some cases but the dynamic configurator is better in general under the dynamic network environment.

VI. Conclusion

In this work, we propose Edgent, an on-demand DNN co-inference framework with device-edge collaboration. Enabling low-latency edge intelligence, Edgent introduces two design knobs to optimize the DNN inference latency: DNN partitioning that enables device-edge collaboration, and DNN right-sizing that leverages early-exit mechanism. We introduce two configurators that are specially designed to figure out the collaboration strategy under static and dynamic bandwidth environments, respectively. Our prototype implementation and the experimental evaluation on Raspberry Pi shows the feasibility and effectiveness of Edgent towards low-latency edge intelligence. For the future work, our proposed framework can be further combined with existing model compression techniques to accelerate DNN inference. Besides, we can extend our framework to support multi-device application scenarios by designing efficient resource allocation algorithms. We hope to stimulate more discussion and efforts in the society and fully realize the vision of edge intelligence.

REFERENCES

[1] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proc. Workshop Mobile Edge Commun. (MECOMM SIGCOMM)*, Budapest, Hungary, Aug. 2018, pp. 31–36.

[2] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1–9.

[3] A. van den Oord *et al.*, "WaveNet: A generative model for raw audio," 2016, *arXiv:1609.03499*. [Online]. Available: https://arxiv.org/abs/1609.03499

[4] D. Wang and E. Nyberg, "A long short-term memory model for answer sentence selection in question answering," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics 7th Int. Joint Conf. Natural Lang. Process.*, vol. 2, 2015, pp. 707–712.

[5] A. Almomani, M. Alauthman, F. Albalas, O. Dorgham, and A. Obeidat, "An online intrusion detection system to cloud computing based on NeuCube algorithms," *Int. J. Cloud Appl. Comput.*, vol. 8, no. 2, pp. 96–112, 2018.

[6] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[8] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[9] Y. Jararweh *et al.*, "Software-defined system support for enabling ubiquitous mobile edge computing," *Comput. J.*, vol. 60, no. 10, pp. 1443–1457, Oct. 2017.

[10] 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./Feb. 2018.

[11] Y. Yang, K. Wang, G. Zhang, X. Chen, X. Luo, and M. Zhou, "MEETS: Maximal energy efficient task scheduling in homogeneous fog networks," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 4076–4087, Oct. 2018.

[12] 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.

[13] X. Chen, W. Li, S. Lu, Z. Zhou, and X. Fu, "Efficient resource allocation for on-demand mobile-edge cloud computing," *IEEE Trans. Veh. Technol.*, vol. 67, no. 9, pp. 8769–8780, Sep. 2018.

[14] C. Xu, J. Ren, D. Zhang, and Y. Zhang, "Distilling at the edge: A local differential privacy obfuscation framework for IoT data analytics," *IEEE Commun. Mag.*, vol. 56, no. 8, pp. 20–25, Aug. 2018.

[15] 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," 2019, *arXiv:1905.10083*. [Online]. Available: https://arxiv.org/abs/1905.10083

[16] *Augmented and Virtual Reality: The First Wave of 5G Killer Apps*, Qualcomm, San Diego, CA, USA, 2017.

[17] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient recurrent neural network execution on mobile GPU," in *Proc. 1st Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2017, pp. 1–6.

[18] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 4820–4828.

[19] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: https://arxiv.org/abs/1602.07360

[20] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proc. ACM Int. Joint Conf. Pervas. Ubiquitous Comput. (UBICOMP)*, Sep. 2015, pp. 283–294.

[21] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: https://arxiv.org/abs/1704.04861

[22] 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: https://arxiv.org/abs/1510.00149

[23] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," 2015, *arXiv:1511.06530*. [Online]. Available: https://arxiv.org/abs/1511.06530

[24] N. D. Lane, S. Bhattacharya, A. Mathur, C. Forlivesi, and F. Kawsar, "DXTK: Enabling resource-efficient deep learning on mobile and embedded devices with the DeepX toolkit," in *Proc. MobiCASE*, 2016, pp. 98–107.

[25] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 2787–2794.

[26] C. Liu *et al.*, "Progressive neural architecture search," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 19–34.

[27] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018, *arXiv:1802.03268*. [Online]. Available: https://arxiv.org/abs/1802.03268

[28] 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, p. 45.

[29] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 65–74.

[30] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 267–278.

[31] A. Graves *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

[32] S. Bateni and C. Liu, "ApNet: Approximation-aware real-time neural network," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2018, pp. 67–79.

[33] B. Islam, Y. Luo, and S. Nirjon, "Zygarde: Time-sensitive on-device deep intelligence on intermittently-powered systems," 2019, *arXiv:1905.03854*. [Online]. Available: https://arxiv.org/abs/1905.03854

[34] 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.

[35] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," 2018, *arXiv:1801.08618*. [Online]. Available: https://arxiv.org/abs/1801.08618

[36] H. Li, K. Ota, and M. Dong, "Learning IoT in edge: Deep learning for the Internet of Things with edge computing," *IEEE Netw.*, vol. 32, no. 1, pp. 96–101, Jan./Feb. 2018.

[37] M. Xu, F. Qian, and S. Pushp, "Enabling cooperative inference of deep learning on wearables and smartphones," 2017, *arXiv:1712.03073*. [Online]. Available: https://arxiv.org/abs/1712.03073

[38] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative IoT devices," 2018, *arXiv:1802.02138*. [Online]. Available: https://arxiv.org/abs/1802.02138

[39] J. Zhou, Y. Wang, K. Ota, and M. Dong, "AAIoT: Accelerating artificial intelligence in IoT systems," *IEEE Wireless Commun. Lett.*, vol. 8, no. 3, pp. 825–828, Jun. 2019.

[40] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, Dec. 2016, pp. 2464–2469.

[41] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: A next-generation open source framework for deep learning," in *Proc. 29th Annu. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1–6.

[42] 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.

[43] Z. Akhtar *et al.*, "Oboe: Auto-tuning video ABR algorithms to network conditions," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, Budapest, Hungary, Aug. 2018, pp. 44–58.

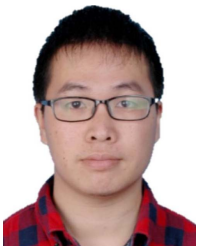
[44] R. P. Adams and D. J. C. MacKay, "Bayesian online changepoint detection," 2007, *arXiv:0710.3742*. [Online]. Available: https://arxiv.org/abs/0710.3742

[45] V. Mulhollon. (2004). *Wondershaper*. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man8/wondershaper.8.html

[46] J. van der Hooft *et al.*, "HTTP/2-based adaptive streaming of HEVC video over 4G/LTE networks," *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2177–2180, Nov. 2016.

[47] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[48] J. Kulick. (2013). *Bayesian Changepoint Detection*. [Online]. Available: https://github.com/hildensia/bayesian_changepoint_detection



En Li received the B.S. degree in communication engineering from the School of Physics and Telecommunication Engineering, South China Normal University (SCNU), Guangzhou, China, in 2017, and the M.S. degree in computer technology from the School of Data and Computer Science, Sun Yat-sen University, Guangzhou. His research interests include mobile deep computing, edge intelligence, and deep learning.

Liekang Zeng received the B.S. degree in computer science from the School of Data and Computer Science, Sun Yat-sen University (SYSU), Guangzhou, China, in 2018, where he is currently pursuing the master's degree. His research interests include mobile edge computing, deep learning, and distributed computing.

Zhi Zhou 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 a Research Fellow with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China. Since 2016, he has been a Visiting Scholar at the University of Gottingen. His research interests include edge computing, cloud computing and distributed systems. He was a sole recipient of the 2018 ACM Wuhan and Hubei Computer Society Doctoral Dissertation Award and a recipient of the Best Paper Award of the IEEE UIC 2018. He is the General Co-Chair of the 2018 International Workshop on Intelligent Cloud Computing and Networking (ICCN).

Xu Chen 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, USA, from 2012 to 2014, and a Humboldt Scholar Fellow with the Institute of Computer Science, University of Goettingen, 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 received the prestigious Humboldt Research Fellowship from the Alexander von Humboldt Foundation of Germany, the 2014 Hong Kong Young Scientist Runner-up Award, the 2016 Thousand Talents Plan Award for Young Professionals of China, 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 the 2010 IEEE International Conference on Intelligence and Security Informatics (ISI), the Best Paper Runner-up Award of the 2014 IEEE International Conference on Computer Communications (INFOCOM), and the Best Paper Award of the 2017 IEEE International Conference on Communications (ICC). He is currently an Area Editor of the IEEE OPEN JOURNAL OF THE COMMUNICATIONS SOCIETY, an Associate Editor of the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, the IEEE INTERNET OF THINGS JOURNAL, and the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS (JSAC) SERIES ON NETWORK SOFTWARIZATION AND ENABLERS.