DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices

Xueyu Hou*, Yongjie Guan*, Tao Han*, Ning Zhang†

*New Jersey Institute of Technology, USA, †Windsor University, Canada

*{xh29, yg274, tao.han}@njit.edu, †{ning.zhang}@uwindsor.ca

Abstract—As the number of edge devices with computing resources (e.g., embedded GPUs, mobile phones, and laptops) increases, recent studies demonstrate that it can be beneficial to collaboratively run convolutional neural network (CNN) inference on more than one edge device. However, these studies make strong assumptions on the devices' conditions, and their application is far from practical. In this work, we propose a general method, called *DistrEdge*, to provide CNN inference distribution strategies in environments with multiple IoT edge devices. By addressing heterogeneity in devices, network conditions, and nonlinear characters of CNN computation, DistrEdge is adaptive to a wide range of cases (e.g., with different network conditions, various device types) using deep reinforcement learning technology. We utilize the latest embedded AI computing devices (e.g., NVIDIA Jetson products) to construct cases of heterogeneous devices' types in the experiment. Based on our evaluations, DistrEdge can properly adjust the distribution strategy according to the devices' computing characters and the network conditions. It achieves 1.1 to 3× speedup compared to state-of-the-art methods.

Index Terms—distributed computing, convolutional neural network, edge computing, deep reinforcement learning

I. INTRODUCTION

With the development of deep learning, various applications are utilizing deep neural networks (DNNs). Convolutional neural networks (CNNs) are one of the most popular types of DNNs, and are applied to a wide range of computer vision tasks like image classification [53, 24, 56], object detection [49, 48], and pose detection [8]. As CNN models have high computational complexity, it is challenging to run CNN model inference for edge devices [26, 73, 11, 67] like mobile phones, surveillance cameras, drones. Studies on CNN compression [69, 22, 25] reduce the computational complexity of a CNN model by sacrificing accuracy. Another popular solution is to offload the whole or partial CNN model to a server [32, 27, 57, 61, 66]. However, their dependency on servers and remote network transmission leads to problems like trustworthiness, security, and privacy [60, 12, 42].

Distributing CNN inference on multiple edge devices in a paralleling manner has been studied in [64, 38, 39, 70, 55, 71, 18, 17, 21, 36, 37, 6] and demonstrate its benefits in reducing inference latency, avoiding overflow of memory/power on devices, and utilizing idle computing resources. Different from distributed CNN training on servers, CNN inference distribution on edge devices shows the following characters (detailed discussion in Section II): (1) much lower throughput between devices; (2) heterogeneous computing devices; (3) heterogeneous network throughput conditions. However, as

will be discussed in Section II, state-of-the-art studies on CNN inference distribution [64, 38, 39, 70, 55, 71, 18, 17, 21, 36, 37, 6] address these characters with strong assumptions and significantly limit their applicable cases in practice. Our method (DistrEdge) addresses these characters with Deep Reinforcement Learning (DRL) and reaches 1.1 to 3× speedup compared to state-of-the-art methods.

The CNN inference distribution method we propose is called *DistrEdge*. Given a group of edge devices and a CNN model for inference, DistrEdge can provide a distribution solution to run inference on these devices by comprehensively analyzing the CNN layer configurations, network conditions, and devices' computing characters. Compared to state-of-the-art studies, we address two practical facts. First, the edge devices available for CNN inference distribution can be heterogeneous in devices' types and network conditions. Second, the relationship between computing latency and CNN layer configurations can be highly nonlinear on edge devices [62], referred to as *nonlinear* character in this paper. One note is that the memory constraint is not considered in this paper because the latest embedded AI computing devices (e.g., NVIDIA Jetson) has adequate memory [43, 5] and the memory consumption by CNN inference does not cause severe issues on them. Overall, the contributions of this paper are as follows:

- As far as we know, DistrEdge is the first CNN inference distribution method that can apply to distribution on heterogeneous edge devices with *nonlinear* characters.
- As far as we know, DistrEdge is the first CNN inference distribution method that models the *split* process as a Markov Decision Process (MDP) and utilizes Deep Reinforcement Learning (DRL) to make optimal split decisions.
- We evaluate the performance of CNN inference on edge devices, including the latest AI computing devices of NVIDIA Jetson products [43]. In comparison, the experiments in state-of-the-art studies limit their test cases with low-speed edge devices like Raspberry Pi3/4 (e.g., [70, 17, 71]) or Nexus 5 (e.g., [38, 39]).
- In the experiment, DistrEdge can achieve 1.1× to 3× speedup over state-of-the-art CNN distribution methods.

II. RELATED WORK

A. Model Parallelism in DNN Training

State-of-the-art model parallelism studies mainly focus on accelerating the training of DNNs on distributed server devices. In general, there are two types of model parallelism in

training DNNs. The first type is graph partitioning [2, 14, 15, 13, 16, 23, 9, 41, 47, 58]. Due to the sequential dependency between the computation tasks in a deep neural network, most graph partitioning works use pipeline parallelism [14, 15, 58]. By splitting training batch into micro-batches, the sequential computation of each micro-batch can be pipelined with those of the other micro-batches [58]. The second type is tensor splitting [51, 52, 31, 30, 59]. The tensor splitting adjusts the input/output data placement on the devices and splits corresponding computation.

Though the model parallelism has been a popular way in implementing DNN training on multiple devices in a distributed manner [13, 16, 23, 9, 41, 47, 58], it is important to note that the devices on which the training is distributed usually refer to computing units like GPUs connected to one server cluster through high-bandwidth connections like NVLinks and Infini-Band [41, 47, 58]. These computing units are controlled by a single operating system on the cluster. The communication throughput between these computing units is >10Gbps [41, 47, 58]. In contrast, distributed edge devices refer to embedded computing boards like NVIDIA Jetson and Raspberry Pi connected through a wireless network. Each computing board has its own operating system, and the data transmission between these boards is through network protocols like TCP/IP. The communication throughput between these boards is usually $1Mbps$ to $1Gbps$, which is much lower than that between computing units on a cluster. Moreover, there is additional I/O reading and writing delay when transmitting data from one edge device to another. In addition, edge devices are usually responsible for DNN inference instead of training. One feature of DNN inference is that the input data usually with a batch size of 1, and inference requests arrive randomly or periodically. Therefore, pipeline parallelism of training batches is not applicable in the distribution of DNN inference on edge devices.

B. CNN Inference on Distributed Edge Devices

Distributed CNN inference studies adopt tensor splitting in model parallelism. By splitting the input/output data (and corresponding computation) of layers in a model, these works demonstrate its benefits in reducing inference latency, avoiding overflow of memory/power on devices, and utilizing idle computing resources. However, state-of-the-art studies limit their applicable cases. [36, 37, 6] limit their implementation on a tiny CNN model like LeNet as they make distribution decisions for each neuron (most popular CNN models consist of a vast amount of neurons). [18, 17, 21] aim at splitting a specific layer in a CNN model equally and allocating the split parts on homogeneous edge devices to avoid memory overflow. For works that can be applied to the distribution of popular CNN models: [64, 38, 39] distribute CNN inference in a layer-by-layer way and split each layer in a CNN model based on the edge devices' computing capability [64, 38, 39] and the network throughput [64]; [70, 55, 71] fuse multiple layers together and split the fused layers to reduce the frequency of data transmission between distributed edge devices. Though

fusing layers reduces transmission latency through a wireless network and I/O reading/wring latency, [70, 55, 71] limit their applicable cases as homogeneous edge devices [70, 55] or devices with linear characters [71].

Overall, state-of-the-art studies implement CNN inference distribution with strong assumptions: (1) DeepThings [70] and DeeperThings [55] assume that the edge devices for distribution are homogeneous and the layer(s) are split equally; (2) CoEdge [64], MoDNN [38], MeDNN [39], and AOFL [71] assume that the relationship between the computing latency and the layer configuration is linear and their ratio can be represented by a value (i.e., computing capability); (3) DeepThings [70], DeeperThings [55], MoDNN [38], and MeDNN [39] do not consider the network conditions of edge devices when making distribution decisions; (4) CoEdge [64] and AOFL [71] assume the transmission latency to be proportional to the value of network throughput and the layer configuration, and calculate the split ratio based on the linear models of the computing and network transmission latency. However, as demonstrated in [62], the relationship between computing latency and layer configurations can be highly nonlinear. In addition, the delay caused by I/O reading/writing should also be accounted for in the transmission latency, and calculating the transmission latency purely by the network throughput can be inaccurate.

This paper develops a novel distribution method, DistrEdge, to break the above limitations of state-of-the-art works. Overall, DistrEdge reaches 1.1 to 3× speedup compared to state-of-the-art distribution solutions in heterogeneous edge devices and network conditions without assuming a linear relationship between the computing/transmission latency and the layer configuration. One note is that, for CNN inference on distributed edge devices, we refer to the study that is distributing existing CNN models *without* any modification of the models' architecture. The benefit of it is that it does not require additional training to recover the accuracy because the accuracy performance is *not* affected in the distribution process.

C. Resource Allocation for CNN Inference

Another related topic for CNN inference on distributed edge devices is to study the resource allocation for CNN inference tasks [40, 10, 50], they either regard one inference process as a black-box [10, 50] or split a CNN layer in the same way [40] as in [38]. RL-PDNN [3] targets improving the overall performance of DNN requests from multiple users, and it does not study model parallelism. As they focus on resource allocation for multiple DNN inference tasks, they are out of the scope of this paper's focus.

D. Architecture Search/Modification for Distributed CNN

Different from the studies of CNN inference on distributed edge devices, [33, 4, 20, 19, 68] modify/search for neural architectures that are suitable for distributing on edge devices. The main focus of such studies is to eliminate data dependency

1098

between edge devices during the inference of CNN models. [33, 4, 19, 68] propose different methods to modify existing CNN models based on: hierarchical grouping of categories for image classification [33]; training several disjoint student models using knowledge distillation [4]; tailoring intermediate connections between neurons to form a low-communication-parallelization version of a model [19]; decomposing the spatial dimension by clipping and quantization [68]. [20] proposes to search for novel neural architecture to provide higher concurrency and distribution opportunities. Though modifying/searching neural architecture for CNN distribution can achieve highly efficient parallel neural architecture, it intrinsically shows the following drawbacks: (1) it takes a long time to modify/search optimal neural architecture and to train the neural network in order to recover the accuracy; (2) The modified neural network cannot achieve the same level of accuracy even after retraining [33, 4, 20, 19, 68].

E. Deep Reinforcement Learning in CNN Model Compression

To tackle the complex configurations and performance of CNN architecture, DRL has been used in CNN model compression and is proven to be effective in finding optimal CNN model compression schemes [25, 1, 65, 54, 63]. The configurations of the current layer [25, 1, 65] and the action of the previous layer [25] are defined as the state in the DRL. The action is the compression decision on the current layer [25, 1, 65, 54, 63]. The reward is defined as a function of the performance (e.g., accuracy) [25, 1, 65, 54, 63]. As far as we know, DistrEdge is the first paper that utilizes DRL to find a CNN inference distribution strategy.

III. DEFINITIONS AND CHALLENGES

A. Terms Definition

For the convenience of the following discussion, we define the following terms: (1) *layer-volume*: a layer-volume refers to one or more sequentially-connected layers in a CNN model (the same concept as fused layers in [70, 55, 71]); (2) *split-part*: a split-part is a partial layer-volume. Each (sub-)layer in a split-part is a partial layer in the layer-volume. The number of (sub-)layers in a split-part is equal to that in the layer-volume; (3) *horizontal partition*: the division of a CNN model into layer-volumes; (4) *vertical split*: the division of a layer-volume into split-parts; (5) *partition location*: the original index (in the CNN model) of the first layer in a layer-volume; (6) *partition scheme*: the set of partition locations in a CNN distribution strategy; (7) *split decision*: a set of coordinate values on the height dimension of the last layer in the layer-volume where the layer-volume is vertically split (We focus on the split on one dimension in this paper and leave splitting on multiple dimensions in the future work); (8) *service requester*: the device with CNN inference service requests; (9) *service providers*: the devices that can provide computing resources for CNN inference.

Fig. 1 shows four examples of CNN inference distribution on two edge devices. Note that the distribution solutions in Fig. 1 (b)(c)(d) are special cases of that in Fig. 1 (a).
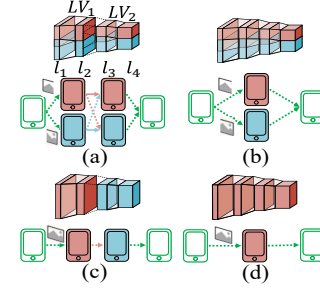

Fig. 1. *CNN inference distribution on two edge devices (blue and red one): (a) a general distribution form; (b) parallel distribution; (c) sequential distribution; (d) single-device offloading.*

Specifically, Fig. 1 (b) takes the whole model as one layer-volume (i.e., parallel distribution on edge devices). Fig. 1 (c) takes each layer-volume as one split-part (i.e., sequentially distribution on edge devices). Fig. 1 (d) takes the whole model as one layer-volume and the layer-volume as one split-part (i.e., offloading to a single device). The design of DistrEdge naturally covers these special distribution forms.

B. Vertical-Splitting Law

A convolutional layer is defined by the following configurations: input width (W_{in}), input height (H_{in}), input depth (C_{in}), output depth (C_{out}), filter size (F), stride (S), padding (P), and an activation function. A maxpooling layer is defined by the following configurations: input width (W_{in}), input height (H_{in}), input depth (C_{in}), filter size (F), and stride (S). The input/output data size and the number of operations of a convolutional/maxpooling layer can be determined given these configurations [17]. For two consecutively connected layers, the dimension of the former layer's output data matches the dimension of the later layer's input data.

For a layer-volume $\tilde{\mathcal{V}}$ with n layers $\{\tilde{l}_1, \tilde{l}_2, ..., \tilde{l}_n\}$ ($n \geq 1$), a split-part p_k is a partial layer-volume with n sub-layers. Each sub-layer is a partial layer of each original layer, e.g., the n-th sub-layer of p_k is a partial layer of $\tilde{l}_n$. As we focus on splitting a layer-volume by the height dimension only, the output width and depth of p_k are the same as those of $\tilde{l}_n$. We denote the output height of p_k as $h_{out}^{n,k}$. The output height of the i-th sub-layer is determined by the height of output data from the $(i+1)$-th sub-layer:

$$h_{out}^{i,k} = (h_{out}^{i+1,k} - 1)S_{i+1} + F_{i+1}, \tag{1}$$

where S_{i+1} and F_{i+1} are the stride and filter size of $\tilde{l}_{i+1}$, respectively. Thus, given $h_{out}^{n,k}$, the output heights the first ($n-1$) sub-layers can be determined by Eq. 1 one by one. Further, the input height of p_k can be calculated by:

$$h_{in}^{1,k} = (h_{out}^{1,k} - 1)S_1 + F_1, \tag{2}$$

The Vertical-Splitting Law is described as:
Vertical-Splitting Law (VSL): For a split-part of a layer-volume with more than one layer, once the output height of its last sub-layer is determined, the input height of its first sub-layer can be calculated by Eq. 1 and Eq. 2.

1099

C. Challenges

We summarize the following challenges in finding an optimal CNN inference distribution strategy on edge devices: *1) High Dimensions of Search Space*: Most CNN models for computer vision tasks have more than ten layers (mainly convolutional layers), and the output size (height) from each layer ranges from less than 10 to over 200 [53, 24, 56, 49, 48]. It is infeasible to brute-force search over all possible combinations of distribution strategies. *2) Heterogeneous and Nonlinear Device Characters*: In practice, the edge devices available for CNN inference distribution can be heterogeneous and with nonlinear characters. The split-parts need to be adjusted accordingly. However, as discussed in Section II, state-of-the-art distribution methods do not fully consider these characters when making their distribution strategies. *3) Various Configurations across Layers in a CNN Model*: The layer configurations (e.g., height, depth, etc.) change across layers in a CNN model, which leads to various performance (i.e., latency). Consequently, the performance of distributed inference is also affected by the layer configurations. *4) Dependency between Layers in CNN Model*: For most CNN models, the layers are connected sequentially one after another [53, 24, 56, 49, 48]. While the operations of the same layer (layer-volume) can be computed in parallel, the operations of two layers (layer-volumes) cannot be computed in parallel. Thus, the devices' executions of the latter layer-volume are affected by the accumulated latencies of the former layer-volume. Consequently, when making splitting decisions to a layer-volume, it is important to consider the accumulated latencies of the split-parts in the former layer-volume.

IV. DISTREDGE

To utilize CNN inference distribution with DistrEdge, a computer (e.g., a laptop) collects information of the service requester and service providers and runs DistrEdge to find an optimal distribution strategy. The information collected includes network throughput between any two devices, layers' configurations of the CNN model, and CNN layer computation/transmission latency profiling on each provider. DistrEdge allows various forms to express the profiling results of a device. It can be regression models (e.g., linear regression, piece-wise linear regression, k-nearest-neighbor) or a measured data table of computing latencies with different layer configurations. Based on the optimal strategy, the controller informs the requester to send the split-parts (i.e., weights and biases) to the corresponding providers. The controller also informs the requester and providers to establish necessary network connections. Afterward, the CNN inference can be offloaded in a distributed manner on these service providers.

A. Workflow of DistrEdge

The workflow of DistrEdge is shown in Fig. 2, which consists of two modules. The first module is a CNN partitioner that partitions a CNN model horizontally. The second module is a layer-volume splitter that splits layer-volumes vertically. The partitioner utilizes the layers' configurations of the CNN

model to find the partition scheme based on the amount of transmission data and the number of operations. Given the partition scheme from the first module, the second module splits the layer-volumes with DRL. By modeling the split process as an MDP, a DRL agent is trained to make optimal split decisions for each layer-volume one by one based on the observations of accumulated latencies and layer configurations (the reward and states). The accumulated latencies implicitly reflect the device characters (i.e., computing capabilities and network conditions). During training, the latencies can be directly measured with real execution on devices or estimated by the profiling results. In this way, DistrEdge can form an inference distribution strategy on service providers.
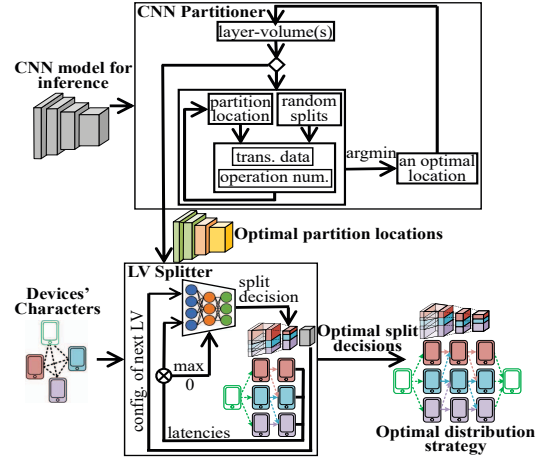

Fig. 2. *Workflow of DistrEdge.*

B. Design of CNN Partitioner

Algorithm 1 shows the proposed partitioner, Layer Configuration based Partition Scheme Search (LC-PSS). As the execution latency is affected by the amount of transmission data and the number of operations in CNN inference distribution [70, 71], we define a score C_p as the function of operation number and transmission amount:

$$C_p = \alpha \cdot \mathcal{T} + (1 - \alpha) \cdot \mathcal{O}. \tag{3}$$

where $\mathcal{O}$ and $\mathcal{T}$ are the total number of operations and the total amount of transmission data in the CNN model, respectively. The hyper-parameter α is used to control the trade-off between the two factors. Given a CNN model $\mathcal{M}$, a partition scheme R_p, and split decisions R_s, $\mathcal{O}$ and $\mathcal{T}$ can be determined. We generate a set of random split decisions $\mathbf{R}_s^r$. With $\mathbf{R}_s^r$, we average the scores of all the $C_p(R_s^i)$ ($R_s^i \in \mathbf{R}_s^r$) given R_p and $\mathcal{M}$, denoted as $\bar{C}_p$. We can formulate the optimal partition scheme search problem as:

$$\min_{R_p} \frac{1}{|\mathbf{R}_s^r|} \sum_{i=1}^{|\mathbf{R}_s^r|} C_p(R_p, R_s^i, \mathcal{M}), R_s^i \in \mathbf{R}_s^r \tag{4}$$

where $|\cdot|$ represents the number of elements (random split decisions) in $\mathbf{R}_s^r$, C_p can be calculated by Eq. 3 given partition locations R_p and split decisions R_s^i of model $\mathcal{M}$. The LC-PSS greedily search for optimal R_p that minimizes Eq. 4.

In the $(k + 1)$-th loop, the LC-PSS searches for an optimal partition location for each layer-volume obtained in the k-th loop (line 4 to 12). The optimal partition locations found in a loop is appended to R_p^* (line 11). The loop stops when no more optimal partition locations are appended to R_p^* (line 12, 13). In the worst case, there will be $(1 + |M|) \cdot |M|/2$ loops to find the optimal partition strategy (i.e., when the optimal partition scheme is to partition the CNN model layer by layer). In contrast, if we brute-forcely search over all possible partition strategies for a CNN model $\mathcal{M}$ [71], there will be $(|M| \cdot |M - 1| \cdot ... \cdot 2)$ loops.

Algorithm 1: Layer Conf. based Partition Scheme Search

Input: $\mathcal{M}$: the CNN model for inference;
        α: the coefficient for partitioning score C_p;
        $\mathbf{R}_s^r$: a set of random split decisions.

1   Initialize partitioning points record $R_p = \{1, |\mathcal{M}|\}$.
2   **while** *True* **do**
3      $R_p^* \leftarrow R_p$;
4   **for** $i = 1, |R_p| - 1$ **do**
5          $C_p^* = \infty$;
6   **for** $j = R_p[i], R_p[i + 1]$ **do**
7              $R_p' = R_p \cup \{j\}$;
8              $\bar{C}_p = \frac{1}{|\mathbf{R}_s^r|} \sum_{i=1}^{|\mathbf{R}_s^r|} C_p(R_p', R_s^i, \mathcal{M}), R_s^i \in \mathbf{R}_s^r$;
9   **if** $\bar{C}_p < C_p^*$ **then**
10                 $C_p^* = \bar{C}_p$; $j^* = j$;
11          $R_p^* = R_p^* \cup \{j^*\}$;
12   **if** $|R_p| = |R_p^*|$ **then**
13   break;
14   **else**
15          $R_p \leftarrow R_p^*$.

Output: R_p^*.

C. Design of LV Splitter

Given service providers $\mathcal{D}$, the CNN model $\mathcal{M}$, and the optimal partition scheme R_p^*, the LV splitter is to make optimal splitting decisions R_s^* that minimizes the end-to-end-execution latency:

$$\min_{R_s} T(\{\mathcal{M}, R_p^*\}; \mathcal{D}; R_s) \tag{5}$$

where R_s is a series of splitting decisions for all layer-volumes $\{\mathcal{V}_1, ..., \mathcal{V}_L\}$ of $\mathcal{M}$, L is the number of layer-volumes in $\mathcal{M}$.

1) Modeling as a Markov Decision Process: The problem of Eq. 5 can be modeled as an MDP. We define the *action* $\boldsymbol{a}_l$ as the splitting parameters for a layer-volume $\mathcal{V}_l$:

$$\boldsymbol{a}_l = (x_1^l, x_2^l, ..., x_{|\mathcal{D}|-1}^l) \tag{6}$$

where $|\mathcal{D}|$ is the total number of service providers, x_i^l ($i \in [1, |\mathcal{D}| - 1]$) is the location on the height-dimension of the last layer in $\mathcal{V}_l$. The locations satisfy $x_i^l \in [0, H^l]$ and $x_i^l \le x_j^l$ when $(i < j)$, where H^l is the height of the last layer in

$\mathcal{V}_l$. The i-th split-part in $\mathcal{V}_l$ is allocated to the i-th service provider. The last sub-layer of the i-th split-part is a sub-layer with operations between x_{i-1}^l and x_i^l on the last layer of $\mathcal{V}_l$ ($x_0^l = 0$ and $x_{|\mathcal{D}|}^l = H^l$). The dimension of the rest sub-layers of the i-th split-part can be determined by Eq. 1. The *state* $\boldsymbol{s}_l$ is defined as:

$$\boldsymbol{s}_l = (\boldsymbol{T}_{l-1}, H^l, C^l, F^l, S^l) \tag{7}$$

where $\boldsymbol{T}_{l-1}$ are accumulated latencies on the service providers after computation of their allocated split-parts of $\mathcal{V}_{l-1}$, (H^l, C^l, F^l, S^l) are configurations of the last layer in $\mathcal{V}_l$ (i.e., height, depth, filter size, and stride).

At step l, the action $\boldsymbol{a}_l$ determines the shapes of the split parts of $\mathcal{V}_l$ allocated to the service providers. On the one hand, the computing latency of a split-part is related to: (1) the shape of its split-part, (2) the device's characters, (3) the layer configurations. On the other hand, the accumulated latencies $\boldsymbol{T}_l$ are related to: (1) the computing latencies of the split parts in $\mathcal{V}_l$, (2) the transmission latencies occurred between (the split-parts of) $\mathcal{V}_{l-1}$ and $\mathcal{V}_l$, (3) the accumulated latencies $\boldsymbol{T}_{l-1}$ for $\mathcal{V}_{l-1}$. Therefore, the accumulated latencies $\boldsymbol{T}_l$ are determined by $\boldsymbol{a}_l$ and $\boldsymbol{s}_l$ together. As $\boldsymbol{T}_l$ are partial of $\boldsymbol{s}_{l+1}$, the splitting problem of Eq. 5 can be regarded as an MDP. In other words, the next state $\boldsymbol{s}_{l+1}$ depends on the current state $\boldsymbol{s}_l$ and the action $\boldsymbol{a}_l$.

2) Splitting Decision Search with DRL: Compared to other optimization algorithms, DRL can deal with the complex environment by learning and can find optimal actions efficiently without an analytical model of the environment. By modeling the split process as an MDP, we can utilize DRL to make optimal split decisions. As we aim at minimizing the end-to-end execution latency T of $\mathcal{M}$, we define the reward r_l as:

$$\begin{cases} r_l = 0 & (l < L), \\ r_l = 1/T & (l = L). \end{cases} \tag{8}$$

For an layer-volume splitter, the splitting parameters (decisions) $(x_1^l, x_2^l, ..., x_{|\mathcal{D}|-1}^l)$ are non-negative integers ranging from 0 to H^l. However, due to this problem's high search space dimension, a discrete action space has an extremely high dimension. Moreover, as the configurations vary across layers in a CNN model, the dimension of a discrete action space changes across layer-volumes. Thus, it is impractical to utilize DRL algorithms with discrete action space in the LV splitter. When applying DRL algorithms with continuous action space, the *action mapping function* is defined:

$$x_i^l = round(H^l \cdot \frac{\tilde{x}_i^l - A}{B - A}), \ (1 \le i \le |\mathcal{D}| - 1) \tag{9}$$

where $round(\cdot)$ is to round a real value to an integer, $\bar{x}_i^l$ is an element in $\boldsymbol{a}_l$, $[A, B]$ is the activation function' boundary in DRL agent. With Eq. 7, Eq. 8, Eq. 6, and Eq. 9, the Optimal Split Decision Search (OSDS) is described in Algorithm 2.

In Algorithm 2, DDPG is an actor-critic DRL algorithm for continuous action space [34]. The DRL agent is trained for Max_{ep} episodes to make optimal policy (split decisions)

Algorithm 2: Optimal Split Decision Search (OSDS)

Input: $\{\mathcal{V}_1, ..., \mathcal{V}_L\}$: layer-volumes of $\mathcal{M}$;

$\quad\quad\quad$ $\mathcal{D}$: service providers;

$\quad\quad\quad$ Max_{ep}: the maximum training episode;

$\quad\quad\quad$ $\Delta\epsilon, \sigma^2$: hyper-parameters for exploration;

$\quad\quad\quad$ N_b, γ: hyper-parameters for training DRL.

1 Randomly initialize critic network $Critic(\boldsymbol{s}, \boldsymbol{a}|\theta_c)$ and actor network $Actor(\boldsymbol{s}|\theta_a)$ with parameters θ_c and θ_a;

2 Initialize target networks $Critic'$ and $Actor'$ with parameters $\theta'_c \leftarrow \theta_c$ and $\theta'_a \leftarrow \theta_a$;

3 Initialize replay buffer $\mathcal{B}$.

4 $T^* = \infty$;

5 for *episode*=1, Max_{ep} **do**

6 $\quad$ Initialize splitting decisions record $R_s = \{\}$;

7 $\quad$ Receive initial observation state $\boldsymbol{s}_1$;

8 $\quad$ $\epsilon = 1 - (episode \cdot \Delta\epsilon)^2$;

9 $\quad$ **for** *h*=1, L **do**

10 $\quad\quad$ **if** $random < \epsilon$ **then**

11 $\quad\quad\quad$ $\tilde{\boldsymbol{a}}_l = Actor(\boldsymbol{s}_l|\theta_a) + \mathcal{N}(0, \sigma^2)$;

12 $\quad\quad$ **else**

13 $\quad\quad\quad$ $\tilde{\boldsymbol{a}}_l = Actor(\boldsymbol{s}_l|\theta_a)$;

14 $\quad\quad$ Map sorted $\tilde{\boldsymbol{a}}_l$ to $\boldsymbol{a}_l$ (Eq. 9);

15 $\quad\quad$ Store $\boldsymbol{a}_l$ into R_s;

16 $\quad\quad$ Split $\mathcal{V}_l$ with $\boldsymbol{a}_l$ and allocate split parts to $\mathcal{D}$;

17 $\quad\quad$ Observe next state $\boldsymbol{s}_{l+1}$ and reward r_l;

18 $\quad\quad$ Store transition $(\boldsymbol{s}_l, \tilde{\boldsymbol{a}}_l, r_l, \boldsymbol{s}_{l+1})$ to $\mathcal{B}$;

19 $\quad\quad$ Sample N_b transitions $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_{i+1})$ from $\mathcal{B}$;

20 $\quad\quad$ Set $y_i = r_i + \gamma Critic'(\boldsymbol{s}_{i+1}, Actor'(\boldsymbol{s}_{i+1}|\theta'_a)|\theta'_c)$;

21 $\quad\quad$ Critic loss: $\frac{1}{N_b}\sum_i(y_i - Critic(\boldsymbol{s}_i, \boldsymbol{a}_i|\theta_c))^2$;

22 $\quad\quad$ Update critic and actor networks.

23 $\quad$ Observe end-to-end execution latency T;

24 $\quad$ **if** $T < T^*$ **then**

25 $\quad\quad$ $R_s^* \leftarrow R_s$; $T^* = T$;

26 $\quad\quad$ $Critic^* \leftarrow Critic$; $Actor^* \leftarrow Actor$.

Output: $R_s^*, Critic^*, Actor^*$.

Fig. 3. *Testbed.*

V. EXPERIMENTS

In this section, we first explain how the testbed is built. Then we demonstrate seven baselines of state-of-the-art CNN inference distribution methods. For the evaluation, we first observe the effect of different α in LC-PSS in different device/network cases. Then we compare the performance (IPS: image-per-second) of DistrEdge with the seven baseline methods in various heterogeneous cases (Table I, II) and large-scale cases (Table III). The hyper-parameters in DistrEdge are set as follows: (1) LC-PSS: $\alpha = 0.75$, $|\mathbf{R}_s^r| = 100$; (2) OSDS: $Max_{ep} = 4000$, $\Delta\epsilon = 1/250$, $\sigma^2 = 0.1$ with four service providers ($\sigma^2 = 1$ with 16 service providers), $N_b = 64$, $\gamma = 0.99$, learning rates of the $Actor$ and $Critic$ network as 10^{-4} and 10^{-3}, respectively. The $Critic$ network is consisted of four fully-connected layers with dimensions of $\{400, 200, 100, 100\}$ and $Actor$ network is consisted of three fully-connected layers with dimensions of $\{400, 200, 100\}$.

A. Test Setup

Our testbed is shown in Fig. 3. Four types (Raspberry Pi3, Nvidia Nano, Nvidia TX2, Nvidia Xavier) of devices are used as service providers. A laptop works as the service controller. A mobile phone is used as the service requester. The wireless router (Linksys AC1900) provides 5GHz WiFi connections among the devices. The router is equipped with OpenWrt [46] system, which allows us to control bandwidth between different devices. The network connections between devices are built with TCP socket. The CNN layers are implemented in TensorRT SDK [44] using FP16 precision and batch size as one. One note is that TensorRT is not a necessary part of the CNN inference distribution implementation. We adopt it to show that DistrEdge is compatible with popular deep learning inference optimizers like TensorRT. For the DRL training in OSDS, we profile the computing latency on each type of device and the transmission latency in each bandwidth level against the height of each layer in a CNN model (granularity as one). Specifically, we use TensorRT Profiler [45] to profile the computing latency. The transmission latency is measured from the time when the data are read from the computing unit (i.e., GPU or CPU) on the sending device to the time when the data are loaded to the memory on the receiving device (both transmission latency and I/O reading/writing latency are included). Each measurement point is repeated 100 times, and we then compute the mean values as the profiled latencies. The profiling results are shared with the service controller, and the controller can utilize the results whenever it needs to run OSDS.

(Line 5). At the l-th step, the DRL agent makes split decisions for $\mathcal{V}_l$ (Line 9), either by random exploration (Line 11) or by the actor network (Line 13). We sort the elements in the original output action vector $\tilde{\boldsymbol{a}}_l$ (Line 11 or 13) from low to high and map the sorted vector to the true splitting parameters $\boldsymbol{a}_l$ with Eq. 9 (Line 14). The layer-volume $\mathcal{V}_l$ is split by $\boldsymbol{a}_l$, and the split-parts are allocated to the service providers in $\mathcal{D}$ (Line 16). Afterward, the next state $\boldsymbol{s}_{l+1}$ and the reward r_l can be observed (Line 17) either by simulation or real execution. In each episode, the end-to-end execution latency T can be measured (Line 23). The optimal splitting decisions R_s^*, T^*, $Critic^*$, and $Actor^*$ are stored (Line 24 to 26). The updating of the critic network and the actor network is illustrated in Line 20 to 22. We still utilize the original output action vector $\tilde{\boldsymbol{a}}_l$ for training the networks (Line 18). N_b tuples are randomly sampled from $\mathcal{B}$ (Line 18, 19) to train the actor-critic networks.

To evaluate the performance of a CNN inference distribution strategy, we stream 5000 images from the service requester to the service providers. An image will not be sent until the result of its previous image is received by the service requester. We measure the overall latency in processing the 5000 images and compute averaged FPS. To reduce overhead, the images are split beforehand according to the distribution strategy. The split-parts on the providers are also preloaded to their memory. On each service provider, three threads are running parallel to implement computation, data receiving, and data transmission by sharing data with a queue. All the codes are implemented in Python3. We mainly focus on the optimal distribution of the convolutional layers and maxpooling layers in this paper. The last fully-connected layer(s) are computed on the service provider with the highest allocated amount of the last layer-volume in our experiments.

B. Baseline Methods

To evaluate the performance of DistrEdge compared to state-of-the-art CNN inference distribution methods, we take seven methods as our baseline: *(1) CoEdge* [64] (linear models for devices and networks, layer-by-layer split); *(2) MoDNN* [38] (linear models for devices, layer-by-layer split); *(3) MeDNN* [39] (linear models for devices, layer-by-layer split); *(4) DeepThings* [70] (equal-split, one fused-layers); *(5) DeeperThings* [55] (equal-split, multiple fused-layers); *(6) AOFL* [71] (linear models for devices and networks, multiple fused layers); *(7) Offload*: We select the service provider with the best computing hardware (e.g., the best GPU) to offload the CNN inference. Note that equal-split refers to splitting layer(s) equally (DeepThings and DeeperThings). In contrast, CoEdge, AOFL, MoDNN, and MeDNN split layer(s) based on linear models of computation and/or transmission. Layer-by-layer refers to making split decisions per layer (CoEdge, MoDNN, and MeDNN). One fused-layer refers to fusing multiple layers once in a CNN model (DeepThings). Multiple fused-layers refer to allowing more than one fused-layers in a CNN model (DeeperThings and AOFL). The term *fused-layer* is equivalent to *layer-volume* in DistrEdge.

C. α and $|\mathbf{R}_s^r|$ in LC-PSS

We first study the effect of the hyper-parameter α and $|\mathbf{R}_s^r|$ in LC-PSS (Algorithm 1), as shown in Fig. 4 and 5.

α controls the trade-off between $\mathcal{O}$ and $\mathcal{T}$ when calculating C_p (Eq. 3). We observe that, when α is smaller, LC-PSS tends to partition the CNN model into more layer-volumes, and vice versa. For example, VGG-16 is partitioned into 16 layer-volumes with $\alpha = 0$ and is partitioned into only two layer-volumes with $\alpha = 1$. To observe the overall performance with different α, we further run the OSDS (Algorithm 2) with the partition schemes obtained with $\alpha = \{0, 0.25, 0.5, 0.75, 1.0\}$ under four types of environments for VGG-16 inference: (1) We have four homogeneous service providers under 200Mbps WiFi; (2) The service providers' types are heterogeneous (Group DB in Table I); (3) The service providers' WiFi bandwidths are heterogeneous (Group NA in Table II); (4)

We have 16 service (Group LB, LC, LD in Table III). The performance comparison is shown in Fig. 4 (a) to (d). We can observe poor performance with $\alpha = 0$ (C_p is only affected by the number of operations) and $\alpha = 1$ (C_p is only affected by the amount of transmission data). The highest performance is achieved with $\alpha = 0.75$ in all cases. Overall, we should consider both the number of operations and the amount of transmission data when searching for the optimal partition scheme. We set $\alpha = 0.75$ in the rest of our experiments. This study also shows that distribution CNN model layer-by-layer [64, 38, 39] is not efficient (e.g., $\alpha = 0$).
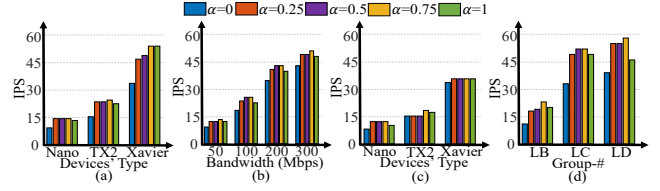

Fig. 4. *IPS w/ different α (VGG-16): (a) homogeneous devices (200Mbp); (b) heterogeneous devices' types (Group DB); (c) heterogeneous network bandwidth (Group NA); (d) large-scale devices.*

$|\mathbf{R}_s^r|$ is the number of random split decisions in Eq. 4. It affects the averaged score $\bar{C}_p$ in Algorithm 1 (line 8). With a small $|\mathbf{R}_s^r|$, the optimal partition locations generated by LC-PSS can change significantly, as different groups of random split decisions lead to different $\bar{C}_p$. With a large $|\mathbf{R}_s^r|$, the optimal partition locations generated by LC-PSS does not show much difference, as different groups of random split decisions lead to similar $\bar{C}_p$. We compare the overall performance (IPS) with different $|\mathbf{R}_s^r|$ in two cases (Group-DB with 50Mbps WiFi and Group-NA with Nano), as shown in Fig 5. For each $|\mathbf{R}_s^r|$ value, we repeatedly run LC-PSS 50 times and record the optimal partition locations of each time. We aggregate the IPS performance over 50 times for each $|\mathbf{R}_s^r|$ value in Fig. 5, and the maximum, average, and minimum IPS among 50 results are shown. When $|\mathbf{R}_s^r|$ is small (e.g., 25 and 50), the IPS varies in a wide range. When $|\mathbf{R}_s^r|$ is large (larger than or equal to 100), the IPS keeps stable. Thus, we set $|\mathbf{R}_s^r|$ to 100 in the rest of our tests.
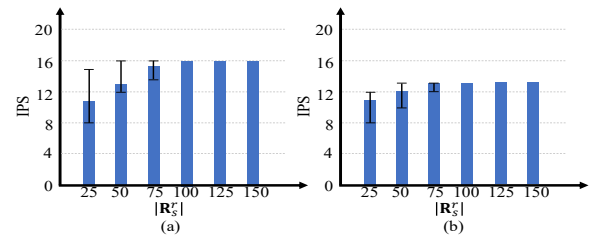

Fig. 5. *Image per Second w/ different number of random split decisions in LC-PSS (VGG-16): (a) DB, 50Mbps WiFi; (b) NA, Nano.*

D. Performance under Heterogeneous Environments

To evaluate the performance of DistrEdge in a heterogeneous environment, we built up multiple cases in the laboratory. These cases can be divided into three types: (1) cases with heterogeneous devices (Table I); (2) cases with heterogeneous network bandwidths (Table II); (3) cases with

large-scale devices (Table III). We utilize image-per-second (IPS) as the performance metric.

TABLE I
GROUPS OF HETEROGENEOUS DEVICES' TYPES

Group #	Devices' Types
DA	TX2×2+Nano×2
DB	Xavier×2+Nano×2
DC	Xavier×1+TX2×1+Nano×1+Pi3×1

TABLE II
GROUPS OF HETEROGENEOUS NETWORK BANDWIDTHS

Group #	Network Bandwidths (Mbps)
NA	50×2+200×2
NB	100×2+200×2
NC	200×2+300×2
ND	50×1+100×1+200×1+ 300×1

1) Heterogeneous Devices: The performance of DistrEdge and baseline methods is shown in Fig. 6. For each group, we set two network bandwidths {50Mbps, 300Mbps}. As shown in Fig. 6, DistrEdge shows higher performance compared to the baseline methods in all cases of heterogeneous devices' types. The performance of DistrEdge shows higher performance improvement over the baseline methods in Group-DB, and DC (e.g., 1.5 to 3× over AOFL) compared to that in Group-DA (e.g., 1.2 to 1.5× over AOFL). As the computing capability comparison of the four devices' types is Pi3<<Nano<TX2<Xavier [29, 28], the cases in Group-DB and Group-DC can be regarded as environments with higher heterogeneous levels compared to those in Group-DA. We can see that DistrEdge is more adaptive to heterogeneous devices' types compared to state-of-the-art methods.
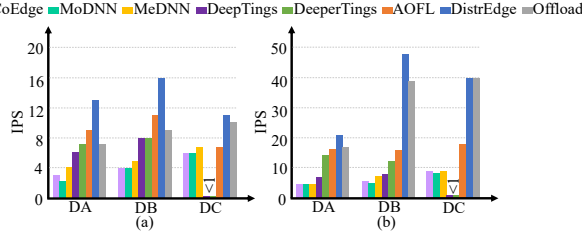
Fig. 6. *Image per Second under Environments w/ Heterogeneous Devices (VGG-16): (a) 50Mbps WiFi; (b) 300Mbps WiFi.*

2) Heterogeneous Network Bandwidths: The performance of DistrEdge and baseline methods is shown in Fig. 7. For each group, we set two devices' types {Nano, Xavier}. As shown in Fig. 7, DistrEdge shows higher performance compared to the baseline methods in all cases of heterogeneous network bandwidths. The performance of DistrEdge shows higher performance improvement over the baseline methods in Group-NA, and ND (e.g., 1.2 to 1.7× over AOFL) compared to that in Group-NB and NC (e.g., 1.1 to 1.3× over AOFL). As shown in Table II, the cases in Group-NA and Group-ND are environments with higher heterogeneous levels compared to those in Group-NB and Group-NC. We can see that DistrEdge is more adaptive to heterogeneous network conditions compared to state-of-the-art methods.

3) Large-Scale Devices: The performance of DistrEdge and baseline methods is shown in Fig. 8. Case-LA is with heterogeneous network bandwidth. Case-LB and Case-LD are

TABLE III
GROUPS OF LARGE-SCALE SERVICE PROVIDERS (16 DEVICES)

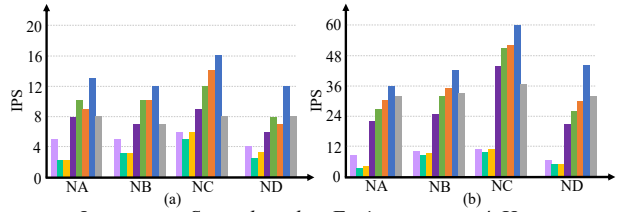| Case #	{(Bandwidths (Mbps), Devices' Types)}
LA	{(300, Nano), (200, Nano), (100, Nano), (50, Nano)} × 4
LB	{(300, Pi3), (200, Nano), (100, TX2), (50, Xavier)} × 4
LC	{(200, Pi3), (200, Nano), (200, TX2), (200, Xavier)} × 4
LD	{(50, Pi3), (100, Nano), (200, TX2), (300, Xavier)} × 4

Fig. 7. *Image per Second under Environments w/ Heterogeneous Networks (VGG-16): (a) Nano; (b) Xavier.*

with heterogeneous network bandwidth and devices' types. Case-LC is with heterogeneous devices' types. As shown in Fig. 8, DistrEdge shows higher performance in the four cases of large-scale service providers compared to the baselines (e.g., 1.3 to 3× over AOFL).
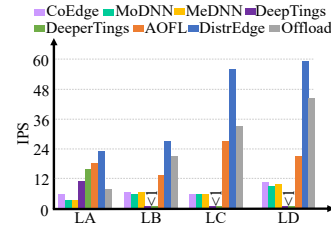

Fig. 8. *Image per Second w/ Large-Scale Devices (VGG-16).*

E. Performance w/ Different Models

Besides VGG-16 (classification), we implement inference distribution for other seven models of different applications: ResNet50 [24] (classification), InceptionV3 [56] (classification), YOLOv2 [48] (object detection), SSD-ResNet50 [35] (object detection), SSD-VGG16 [35] (object detection), Open-Pose [7] (pose detection), VoxelNet [72] (3D object detection). We test under two cases: (1) Group-DB (Table I) with 50Mbps WiFi, as shown in Fig. 9; (2) Group-NA (Table II) with Nano, as shown in Fig. 10. As shown in Fig. 9 and Fig. 10, DistrEdge outperforms the baseline methods in all the models by 1.1× to over 2.6×.
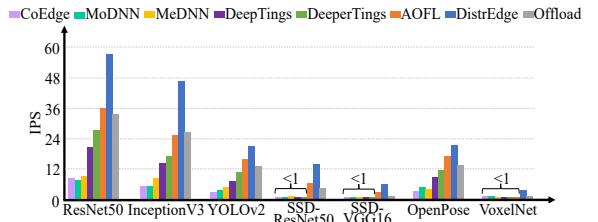

Fig. 9. *Image per Second w/ Different Models (DB, 50Mbps).*

F. Performance in Highly Dynamic Networks

The above results are in the WiFi network which has small fluctuation. We further evaluate the performance of DistrEdge
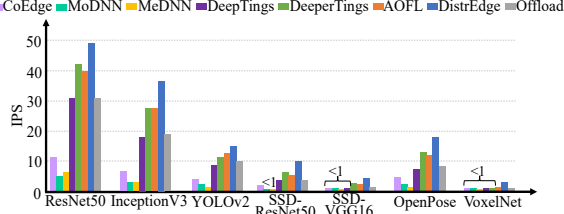
Fig. 10. *Image per Second w/ Different Models (NA, Nano).*

Fig. 12. *Per-Image Processing Latency (ms) w/ Different Methods.*

under highly dynamic network conditions. Specifically, we test with distribution on four devices (Nano) under highly dynamic network conditions. We generate four network throughput traces with high fluctuations shown in Fig. 11. Among the baseline methods, CoEdge [64] and AOFL [71] are applicable to dynamic networks. We compare the performance of DistrEdge with CoEdge and AOFL, as shown in Fig. 12. The split decisions of all the three methods are made on a controller (ThinkPad L13 Gen 2 Intel) in an online manner based on the monitored network throughput (CoEdge and AOFL) and intermediate latency (DistrEdge). Note that the trained actor network of DistrEdge runs on the controller to make online split decisions. For DistrEdge and AOFL, the model partition locations are updated when averaged network throughput is detected to be changed significantly (e.g., at 20min and 40min time-slot in Fig. 11). For DistrEdge, the actor network is finetuned on the controller after partition location adjustment, which takes 20s to 210s. For AOFL, due to brute-force search for optimal partition locations, it takes 10min to update distribution scheme on the same controller.
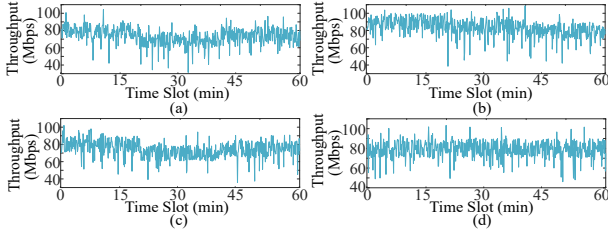

Fig. 11. *Sampled Network Throughput (Mbps) of Highly Dynamic Network: (a) Device 1; (b) Device 2; (c) Device 3; (d) Device 4.*

As shown in Fig. 12, the per-image processing latency of CoEdge [64] is the largest because it does not utilize layer-volumes to fuse layers. As data transmission occurs between any two connected layers, large transmission latency is generated, increasing the end-to-end execution latency of the model. In contrast, both AOFL [71] and DistrEdge reduce the transmission latency by fusing layers into layer-volumes. The key difference between AOFL and DistrEdge is that: AOFL makes split decisions based on a linear ratio, but DistrEdge makes split decisions based on the intermediate latency and learned relationship between latency and split decisions. As shown in Fig. 12, the per-image processing latency of DistrEdge is 40% to 65% of that of AOFL.

G. Why DistrEdge Outperforms Baselines

For CoEdge [64], MoDNN [38], and MeDNN [39], as they split a CNN model layer-by-layer, data transmission through
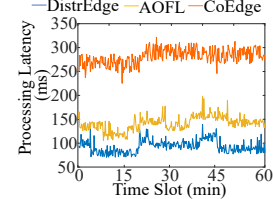
networks occurs between any two connected layers. Thus, they show large end-to-end latency due to transmission delay. For DeepThings [70] and DeeperThings [55], though they fuse layers to reduce transmission delay, they only consider equal-split of fused layers (layer-volume) by assuming that the devices are homogeneous. For AOFL [71], it split layer-volumes based on a linear ratio. However, as shown in Fig. 13, when the relationship between computing latency and layer configuration is nonlinear, splitting layer-volumes with a linear ratio cannot balance the computing latency. Instead, it can cause computing delay. For *offload*, as it only utilizes one device to compute the CNN model, the end-to-end latency can be larger than the distribution methods that compute the layers in a parallel manner on multiple devices.
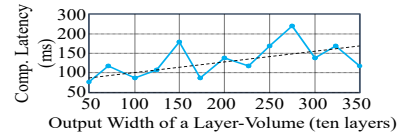

Fig. 13. *Computing Latency v.s. Output Width in CNN.*

We demonstrate the maximum transmission latency and the maximum computing latency among the four devices of Group-DB (50Mbps WiFi) with different distribution methods. As discussed above, distributions with CoEdge [64], MoDNN [38], and MeDNN [39] have large transmission latency due to layer-to-layer data transmission. DeepThings [70] and DeeperThings [55] have large computing latency because they equally split layer-volumes among devices. As the computing capability of Nano is much smaller than that of Xavier, equally splitting causes high computing delay. AOFL [71] splits layer-volumes with a linear ratio, which can also cause computing delays on devices. In comparison, DistrEdge implicitly learns the nonlinear relationship between the computing latency and layer configurations with DRL. It makes split decisions based on the observations of intermediate latency of previous layer-volumes and minimizes the expected end-to-end latency.
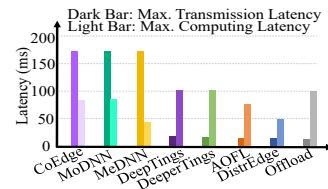

Fig. 14. *Max. Transmission Latency (ms) and Max. Computing Latency (ms) among Four Devices (DB, 50Mbps) w/ Different Methods.*

VI. DISCUSSION

We address the following points in the discussion:

(1) State-of-the-art methods (CoEdge and AOFL) that are applicable highly dynamic network environment makes online split decisions based on the monitored network throughput. Similarly, DistrEdge can also make online split decisions (as discussed in Section V-F) by keeping the actor network online. When network condition changes significantly, AOFL and DistrEdge all need to update their distribution strategy (including partition locations). Benefiting from the lightweight LC-PSS, DistrEdge can update its distribution strategy faster than AOFL based on our evaluation (Section V-F).

(2) Based on the network and edge devices' conditions, DistrEdge divides a CNN model into several split parts. Then it allocates these parts to the edge devices and establishes necessary connections among them. When a user requests CNN inference services, DistrEdge tells the user how to split their input images and which edge devices to send the split images. After the corresponding edge devices generate results, they send the results back to the user. Note that it is possible to obtain an empty set for an edge device. For example, the Pi3 in Group-DC (Table I) is not allocated with any computation due to its low computing capability.

(3) TensorRT is one of the popular deep learning acceleration frameworks provided by NVIDIA. We utilize it in our tests to demonstrate that DistrEdge is compatible with other deep learning acceleration techniques. In general, DistrEdge can be utilized alone without any acceleration technique or can be utilized together which other acceleration techniques.

(4) We do not consider memory constraint in DistrEdge because state-of-the-art edge devices can provide sufficient memory space for most state-of-the-art CNN inferences [5]. Specifically, the state-of-the-art CNN models consume less than 1.5GB of memory. In contrast, the state-of-the-art edge devices are equipped with the memory size of 4GB to 32GB. In other words, even running a whole CNN model on one edge device does not suffer from memory limitation.

(5) In this paper, we study DistrEdge with one dimension split. In the future, we will explore multi-dimension split.

VII. CONCLUSION

In this paper, we proposed DistrEdge, an efficient CNN inference method on distributed edge devices. DistrEdge consists of a CNN partitioner (LC-PSS) and an LV splitter (OSDS) to find CNN inference distribution strategy. We evaluated DistrEdge in laboratory environments with four types of embedded AI computing devices, including NVIDIA Jetson products. Based on our evaluations, we showed that DistrEdge is adaptive to various cases with heterogeneous edge devices, different network conditions, small/large-scale edge devices, and different CNN models. Overall, we observed that DistrEdge achieves 1.1 to $3\times$ speedup compared to the best-performance baseline method.

ACKNOWLEDGEMENT

REFERENCES

[1] Anubhav Ashok et al. "N2n learning: Network to network compression via policy gradient reinforcement learning". In: *arXiv preprint arXiv:1709.06030* (2017).

[2] Guy B. et al. "Pipelining with futures". In: *TCS* (1999).

[3] Emna Baccour et al. "RL-PDNN: Reinforcement Learning for Privacy-Aware Distributed Neural Networks in IoT Systems". In: *IEEE Access* 9 (2021), pp. 54872–54887.

[4] K. Bhardwaj et al. "Memory and communication aware model compression for distributed deep learning inference on iot". In: *TECS* (2019).

[5] Simone Bianco et al. "Benchmark analysis of representative deep neural network architectures". In: *IEEE Access* 6 (2018), pp. 64270–64277.

[6] Elias C. et al. "Dianne: Distributed artificial neural networks for the internet of things". In: *WMCAAI*. 2015.

[7] Zhe Cao et al. "OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields". In: *IEEE transactions on pattern analysis and machine intelligence* 43.1 (2019), pp. 172–186.

[8] Zhe Cao et al. "Realtime multi-person 2d pose estimation using part affinity fields". In: *IEEE CVPR*. 2017.

[9] Chi-Chung Chen et al. "Efficient and robust parallel dnn training through model parallelism on multi-gpu platform". In: *arXiv preprint arXiv:1809.02839* (2018).

[10] Jienan Chen et al. "iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks". In: *ITJ* (2019).

[11] Shichao Chen et al. "Recent advances in collaborative scheduling of computing tasks in an edge computing paradigm". In: *Sensors* (2021).

[12] B. Enkhtaivan et al. "Mediating data trustworthiness by using trusted hardware between iot devices and blockchain". In: *SmartIoT*. IEEE. 2020.

[13] Alexander L Gaunt et al. "AMPNet: Asynchronous model-parallel training for dynamic neural networks". In: *arXiv preprint arXiv:1705.09786* (2017).

[14] John Giacomoni et al. "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue". In: *ACM SIGPLAN*. 2008.

[15] Michael I Gordon et al. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs". In: *ACM SIGPLAN Notices* (2006).

[16] Lei Guan et al. "XPipe: Efficient pipeline model parallelism for multi-GPU DNN training". In: *arXiv preprint arXiv:1911.04610* (2019).

[17] Ramyad H. et al. "Collaborative execution of deep neural networks on internet of things devices". In: *arXiv preprint arXiv:1901.02537* (2019).

[18] Ramyad H. et al. "Distributed perception by collaborative robots". In: *RAL* (2018).

[19] Ramyad H. et al. "LCP: A Low-Communication Parallelization Method for Fast Neural Network Inference in Image Recognition". In: *arXiv preprint arXiv:2003.06464* (2020).

[20] Ramyad H. et al. "Reducing Inference Latency with Concurrent Architectures for Image Recognition". In: *arXiv preprint arXiv:2011.07092* (2020).

[21] Ramyad H. et al. "Toward collaborative inferencing of deep neural networks on Internet-of-Things devices". In: *ITJ* (2020).

[22] Song Han et al. "Learning both weights and connections for efficient neural networks". In: *arXiv preprint arXiv:1506.02626* (2015).

[23] Aaron Harlap et al. "Pipedream: Fast and efficient pipeline parallel dnn training". In: *arXiv preprint arXiv:1806.03377* (2018).

[24] Kaiming He et al. "Deep residual learning for image recognition". In: *IEEE CVPR*. 2016.

[25] Yihui He et al. "Amc: Automl for model compression and acceleration on mobile devices". In: *ECCV*. 2018.

[26] Xueyu Hou and Tao Han. "TrustServing: A Quality Inspection Sampling Approach for Remote DNN Services". In: *2020 17th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. 2020.

[27] Hyuk-Jin Jeong et al. "IONN: Incremental offloading of neural network computations from mobile devices to edge servers". In: *SCC*. 2018.

[28] *Jetson Benchmarks*. https://developer.nvidia.com/embedded/jetson-benchmarks. Accessed: 2021-06-04.

[29] *Jetson Nano: Deep Learning Inference Benchmarks*. https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks. Accessed: 2021-06-04.

[30] Zhihao Jia et al. "Beyond data and model parallelism for deep neural networks". In: *arXiv preprint arXiv:1807.05358* (2018).

[31] Zhihao Jia et al. "Exploring hidden dimensions in parallelizing convolutional neural networks". In: *arXiv preprint arXiv:1802.04924* (2018).

[32] Yiping Kang et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge". In: *ACM SIGARCH Computer Architecture News* (2017).

[33] Juyong Kim et al. "Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization". In: *ICML*. 2017.

[34] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[35] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[36] Fabiola M. et al. "Partitioning convolutional neural networks for inference on constrained internet-of-things devices". In: *30th SBAC-PAD*. IEEE. 2018.

[37] Fabola M. et al. "Partitioning convolutional neural networks to maximize the inference rate on constrained IoT devices". In: *Future Internet* (2019).

[38] J. Mao et al. "Modnn: Local distributed mobile computing system for deep neural network". In: *DATE*. 2017.

[39] Jiachen Mao et al. "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns". In: *ICCAD*. IEEE. 2017.

[40] Thaha Mohammed et al. "Distributed inference acceleration with adaptive DNN partitioning and offloading". In: *INFOCOM*. IEEE. 2020.

[41] Deepak Narayanan et al. "Memory-efficient pipeline-parallel dnn training". In: *ICML*. 2021.

[42] Anggi Pratama Nasution et al. "IoT Object Security towards On-off Attack Using Trustworthiness Management". In: *8th ICoICT*. IEEE. 2020.

[43] *NVIDIA Jetson Products*. https://developer.nvidia.com/buy-jetson. Accessed: 2021-06-02.

[44] *NVIDIA TensorRT*. https://developer.nvidia.com/tensorrt. Accessed: 2021-05-31.

[45] *NVIDIA TensorRT Profiler*. https://docs.nvidia.com/deeplearning/tensorrt/best-practices/index.html#profiling. Accessed: 2021-05-31.

[46] *OpenWrt*. https://openwrt.org/. Accessed: 2021-06-04.

[47] Jay Park et al. "Enabling large DNN training on heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism". In: *ATC*. 2020.

[48] Joseph Redmon et al. "YOLO9000: Better, Faster, Stronger". In: *arXiv preprint arXiv:1612.08242* (2016).

[49] Shaoqing Ren et al. "Towards real-time object detection with region proposal networks". In: *NIPS* (2015).

[50] Yuvraj Sahni et al. "Data-aware task allocation for achieving low latency in collaborative edge computing". In: *ITJ* (2018).

[51] Noam Shazeer et al. "Deep learning for supercomputers". In: *arXiv preprint arXiv:1811.02084* (2018).

[52] Mohammad Shoeybi et al. "Training multi-billion parameter language models using model parallelism". In: *arXiv preprint arXiv:1909.08053* (2019).

[53] Karen Simonyan et al. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[54] Ravi Soni et al. "HMC: A Hybrid Reinforcement Learning Based Model Compression for Healthcare Applications". In: *15th CASE*. IEEE. 2019.

[55] Rafael Stahl et al. "DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices". In: *IJPP* (2021).

[56] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *IEEE CVPR*. 2016.

[57] Surat T. et al. "Distributed deep neural networks over the cloud, the edge and end devices". In: *ICDCS*. 2017.

[58] Masahiro Tanaka et al. "Automatic Graph Partitioning for Very Large-scale Deep Learning". In: *arXiv preprint arXiv:2103.16063* (2021).

[59] Minjie Wang et al. "Supporting very large models using automatic dataflow graph partitioning". In: *EuroSys'19*.

[60] Haiqin Wu et al. "Enabling data trustworthiness and user privacy in mobile crowdsensing". In: *IEEE/ACM Transactions on Networking* (2019).

[61] Shuochao Yao et al. "Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency". In: *SenSys*. 2020.

[62] Shuochao Yao et al. "Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices". In: *16th SenSys*. 2018.

[63] Sixing Yu et al. "GNN-RL Compression: Topology-Aware Network Pruning using Multi-stage Graph Embedding and Reinforcement Learning". In: *arXiv preprint arXiv:2102.03214* (2021).

[64] Liekang Zeng et al. "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices". In: *Transactions on Networking* (2020).

[65] Huixin Zhan et al. "Deep Model Compression via Two-Stage Deep Reinforcement Learning". In: *MLKDD'21*.

[66] Beibei Zhang et al. "Dynamic DNN Decomposition for Lossless Synergistic Inference". In: *arXiv preprint arXiv:2101.05952* (2021).

[67] PeiYun Zhang, MengChu Zhou, and Giancarlo Fortino. "Security and trust issues in fog computing: A survey". In: *Future Generation Computer Systems* (2018).

[68] Sai Qian Zhang et al. "Adaptive Distributed Convolutional Neural Network Inference at the Network Edge with AD-CNN". In: *49th ICPP*. 2020.

[69] Xiangyu Zhang et al. "Accelerating very deep convolutional networks for classification and detection". In: *transactions on PAMI* (2015).

[70] Zhuoran Zhao et al. "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters". In: *Transactions on CADICS* (2018).

[71] Li Zhou et al. "Adaptive parallel execution of deep neural networks on heterogeneous edge devices". In: *4th SEC*. 2019.

[72] Yin Zhou and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4490–4499.

[73] Xiaojian Zhu and MengChu Zhou. "Multiobjective Optimized Cloudlet Deployment and Task Offloading for Mobile-Edge Computing". In: *IEEE Internet of Things Journal* (2021).