Fast and Accurate Streaming CNN Inference via Communication Compression on the Edge

Diyi Hu
University of Southern California
diyihu@usc.edu

Bhaskar Krishnamachari
University of Southern California
bkrishna@usc.edu

***Abstract*—Recently, compact CNN models have been developed to enable computer vision on the edge. While the small model size reduces the storage overhead and the light-weight layer operations alleviate the burden of the edge processors, it is still challenging to sustain high inference performance due to limited and varying inter-device bandwidth. We propose a streaming inference framework to simultaneously improve throughput and accuracy by communication compression. Specifically, we perform the following optimizations: 1) *Partition*: we split the CNN layers such that the devices achieve computation load-balance; 2) *Compression*: we identify inter-device communication bottlenecks and insert Auto-Encoders into the original CNN to compress data traffic; 3) *Scheduling*: we adaptively select the compression ratio when the variation of bandwidth is large. The above optimizations improve inference throughput significantly due to better communication performance. More importantly, accuracy also increases since 1) fewer frames are dropped when input images are streamed in at a high rate, and 2) the frames successfully entering the pipeline are processed accurately since the AE-based compression incurs negligible information loss. We evaluate MobileNet-v2 on pipeline of Raspberry Pi 3B+. Our compression techniques lead to up to 32% accuracy improvement, when average Wi-Fi bandwidth varies from 3 to 9Mbps.**

***Index Terms*—Edge computing, CNN, Data Compression**

I. Introduction

Convolutional Neural Networks (CNNs) are fundamental models for Computer Vision. Conventionally, CNN inference is performed on the cloud, while input data is collected on the edge. Unfortunately, such cloud-centric paradigm requires long distance data transmission, resulting in substantial upload bandwidth consumption, high latency and privacy concerns [1]. Thus, a recent trend is inference on the edge.

To close the natural gap between complex CNN models and resource-constrained edge devices, researchers have designed compact CNNs [2]–[6]. Keeping accuracy unaffected, these models relieve the memory storage pressure of edge devices with their small model sizes, and alleviate the burden on the edge processors with their light-weight layer operations.

In applications such as vehicle detection and video analytics, *streaming* input data are collected by IoT sensors and continuously generated at high rate. However, it is non-trivial to optimize inference throughput for streaming data, due to:

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

a) Bandwidth scarcity: To improve throughput, a widely used approach, model partitioning, splits the CNN into multiple groups of layers. We then deploy each group to an edge device or edge server. However, limited network bandwidth hinders performance of such deployment. The works of [7] [8] replace the original CNN with a smaller one using early-exit or distillation [9]. However, these techniques are not ideal for 2 reasons. First, emerging CNN models such as MobileNetV2 are already very compact and hard to compress. Second, both techniques aim at shrinking the CNN model. The reduction to the hidden layer output to be communicated is limited.

b) Bandwidth variation: Due to interference and varying signal strength, the network channel bandwidth is varying over time. Works in [7] [10] show that CNN inference performance is highly sensitive to the change of bandwidth. It remains a question how to dynamically adjust the inference framework according to the bandwidth. For example, when bandwidth is low, it is desirable to have less intermediate data transmission to avoid throughput degradation. On the other hand, such restriction on data transmission may lower inference accuracy.

We design a CNN inference framework on the local edge device cluster. Following model partition, we assign grouped layers of CNN to edge devices that form a pipeline. We address the above two challenges by compressing intermediate activations that are communicated between edge devices. We use AutoEncoder (AE) to achieve high compression ratio. End-to-end training is performed on the compressed CNN for accuracy recovery. Our main contributions are as follows:

- We propose a composite metric *effective accuracy*, that jointly evaluates throughput and accuracy quantitatively.
- We propose a fast CNN partitioning algorithm to achieve optimal computation load-balance across edge devices.
- We propose data compressors based on the Auto-Encoder architecture to address the communication bottleneck. The compressor is flexible in terms of compression rates, and preserves accuracy with negligible overhead.
- We propose a runtime scheduler that dynamically selects pre-trained compressors per available network bandwidth to optimize inference throughput and effective accuracy.
- We evaluate MobileNet-v2 and on processing pipelines consisting of Raspberry Pi 3B+. Our framework consistently achieves significant accuracy improvement under a wide range of Wi-Fi network bandwidth.

978-1-7281-6602-5/20/$31.00 ©2020 IEEE
DOI 10.1109/IoTDI49375.2020.00023

II. Background and Related Work

A. Compact CNN Models on the Edge

Input (e.g. image stream) to CV applications are collected by cameras on edge devices. Many applications such as smart city, smart grid and Virtual Reality expect data to be analyzed locally on the resource-constrained edge devices [10]. Thus, compact CNNs have been designed for edge inference. SqueezeNet [2] downsamples data using 1×1 convolution filters. It achieves the same accuracy as AlexNet [11] with $50\times$ fewer parameters and only 0.5MB model size. Using customized architecture, YOLO [3] performs accurate real-time object detection with only $\frac{1}{4}$ operations of VGG16 [12].

MobileNet [4]–[6] is one of the state-of-the-art CNN models recently proposed for edge device execution. The core building block is *"inverted residual block"*, which decomposes a regular convolution layer into 3 operations: 1×1 pointwise 2D convolution to expand the number of input channels with ratio t, 3×3 depthwise separable 2D convolution with non-linearity, and 1×1 pointwise 2D convolution to project back the activation to low-dimension. The depthwise separable convolution layer applies a single convolutional filter per input channel. A conventional $k \times k$ conv2d layer with c_{in} input and c_{out} output channels needs c_{out} filters of size $k \times k \times c_{\text{in}}$, while depthwise separable conv2d only needs $c_{\text{in}} \cdot t$ kernels of size $k \times k \times 1$. The number of operations is reduced by $1/k^2$.

Compared with traditional CNN models (e.g. [11], [12]), MobileNet significantly reduces computation complexity without accuracy loss. For example, for the ImageNet [13] dataset consisting of $224 \times 224 \times 3$ images, MobileNetV2 [5] achieves 71.8% classification accuracy in 3.47M model size with only 600M FLOPs. The inference time on a Google Pixel 1 smartphone is only 73.8ms. We will utilize the "inverted residual block" in our compressors (Section IV-C).

B. Related Work

a) CNN compression: Compression reduces CNN model size and computation workload. The compressed model better fits edge devices with limited processing power and memory storage. Works in [14] [15] [8] [16] [17] have accelerated edge inference by model compression. [16] employs *networking pruning* to trim connections having little influence on the inference accuracy. It then uses *data quantization* to reduce the number of bits to represent each model weight value. [8] partitions the CNN model into "head" and "tail". Then the authors use *knowledge distillation* [9] to compress the "head" on edge device, i.e., training a compact "student model" that imitates the behavior of the original "head" ("teacher model").

The above have their limitations. First, emerging models such as MobileNetV2 are already very compact and hard to compress significantly. More importantly, model compression does not directly address the communication bottleneck since the size of the hidden layer outputs is not necessarily reduced.

b) Dynamic schedule: Another direction is to dynamically schedule the inference computation given varying resources. DeepThings [18] partitions a convolution layer into tiles that are distributed among edge devices. Its scheduler performs work-stealing for runtime workload-balance. Edgent [7] trains models with multiple exit points in the later stages. Based on observed bandwidth, they greedily search the best model partition point and exit point. Our work differs from the above. Compared with [18], in addition to load balance, we also consider the dynamics in bandwidth, and adaptively compress the intermediate layer activation transmitted among edge devices. Compared with [7], our approach compresses the communication data, and works well for models with large activation size in early stages, which is the general case [8].

III. Problem Definition

Given an edge device pipeline executing a CNN model, we aim at improving the inference *throughput* and *effective accuracy* when input images are generated at a steady rate.

Optimization goal Suppose the CNN is used for classification, where the image generation rate T_{gen} is a constant and the CNN pre-trained accuracy is ν. *Throughput*, T, is defined as number of classified images per unit time, where $T \leq T_{\text{gen}}$. *Effective accuracy*, $\nu_{\text{eff}} \coloneqq \frac{T \cdot \nu}{T_{\text{gen}}}$, is defined as the ratio of number of correctly classified images over total number of generated images, in unit time. For real-time inference, a dropped frame is equivalent to the image being classified incorrectly. We observe from the ν_{eff} definition a tradeoff between throughput and pre-trained accuracy ν. To improve ν_{eff}, we may compress a CNN such that the decreased ν is compensated by the increased throughput T. This motivates the adaptive compression scheme in Section IV.

The system consists of a linear pipeline of heterogeneous edge devices. Denote C_i as the computation speed of the i^{th} device, and B_i as the bandwidth to transfer data from the i^{th} to the $(i+1)^{\text{th}}$ device. Since each device is mostly executing the same type of convolution operation, we assume C_i remains fixed during inference. However, we may have $C_i \neq C_j$ for $i \neq j$. Regarding bandwidth, we assume the devices communicate over wireless channels and the environment coherence time is large. So B_i changes slowly over time, yet its variance may be significant. Mathematically, we model B_i as *independent random variables*, each following distribution B_i. Define $\mathcal{C} = \{C_i \mid 1 \leq i \leq n\}$ and $\mathcal{B} = \{\mathsf{B}_j \mid 1 \leq j \leq n-1\}$.

Remark on notation For a random variable, we use lower case letter (e.g., b, τ) to denote its value and the Sans Serif font (e.g., B) to denote the probability distribution. Subscript i denotes parameters of device i, or between devices i, $i+1$.

IV. Optimized Pipeline Execution

A natural way to pipeline CNN inference is to split the layers onto the edge devices. Let n and m be the total number of edge devices and CNN layers. Suppose we split the m layers into n parts and the layer indices at the split points are $\mathcal{S} = \{s_1, \ldots, s_{n-1}\}$. For ease of notation, we set $s_0 = 0$ and $s_n = m$. Thus, device i executes layer $s_{i-1}+1$ to layer s_i.

The pipeline under the above configuration consists of n computation stages corresponding to the n devices, and $n-1$ communication stages to transfer layer activation between

adjacent devices. To improve the overall throughput, we have to reduce execution time of the *bottleneck pipeline stage*. Irrespective of the communication stage performance, the overall throughput is bounded by:

$$T \leq \min_{1\leq i\leq n}\left\{\frac{C_i}{\sum\limits_{s_{i-1}+1\leq j\leq s_i}\texttt{ops}_{\mathbb{A}}\left(j\right)}\right\} \leq \frac{\sum\limits_{1\leq i\leq n}C_i}{\sum\limits_{1\leq j\leq m}\texttt{ops}_{\mathbb{A}}\left(j\right)} \tag{1}$$

where $\texttt{ops}_{\mathbb{A}}\left(i\right)$ returns the number of computation operations of layer i. The first inequality is achieved if the $n-1$ communication stages are not the bottleneck, and the second inequality is achieved if $\frac{C_i}{\sum_{s_{i-1}+1\leq j\leq s_i}\texttt{ops}_{\mathbb{A}}(j)} = \frac{C_k}{\sum_{s_{k-1}+1\leq \ell\leq s_k}\texttt{ops}_{\mathbb{A}}(\ell)}$. For given CNN and edge devices, the bound of T is a constant. Thus, to maximize throughput, we need to 1) balance the load of the computation stages (Section IV-A), and 2) reduce the load of the communication stages (Section IV-C).

A. Load-Balance of the Computation Stages

The optimal split points $\mathcal{S}$ can be identified by dynamic programming. Define $\texttt{split}_{\mathbb{A},\mathcal{C}}\left(p,q\right)$ as optimally splitting the last p CNN layers (i.e., layer $m-p+1$ to layer m) onto the last q devices of the pipeline (i.e., device $n-q+1$ to device n). Assume communication stages are not the bottleneck, and let $\texttt{split}_{\mathbb{A},\mathcal{C}}\left(p,q\right)$ return the computation throughput of the q-device pipeline after splitting. We solve $\texttt{split}_{\mathbb{A},\mathcal{C}}\left(m,n\right)$ by:

$$\texttt{split}_{\mathbb{A},\mathcal{C}}\left(p,q\right) = \max_{m-p<s<m}\min\left\{\frac{C_{n-q+1}}{\sum\limits_{k=m-p+1}^{s}\texttt{ops}_{\mathbb{A}}\left(k\right)}, \texttt{split}_{\mathbb{A},\mathcal{C}}\left(m-s,q-1\right)\right\} \tag{2}$$

with the initial condition $\texttt{split}_{\mathbb{A},\mathcal{C}}\left(i,1\right) = \frac{C_n}{\sum_{k=m-i+1}^{m}\texttt{ops}_{\mathbb{A}}(k)}$.

Clearly, we obtain $\texttt{split}_{\mathbb{A},\mathcal{C}}\left(m,n\right)$ by filling a $m\times n$ table. In summary, load-balance of the computation stages can be achieved by the splitting algorithm of complexity $\mathcal{O}\left(m^2n\right)$.

We use T_{p} to denote the throughput of the bottleneck computation stage after the splitting. i.e., $T_{\text{p}} = \texttt{split}_{\mathbb{A},\mathcal{C}}\left(m,n\right)$.

B. Inter-device Communication Bottleneck

After layer splitting to achieve the optimal computation load-balance, limited bandwidth often makes the inter-device communication the bottleneck. Figure 1 visualizes the time taken by each pipeline stage under various configuration. We execute MobileNet-v2 on pipelines of Raspberry Pi and VGG16 on pipelines of NVIDIA Jetson, where inputs are 32×32 R.G.B. images. Here we simplify the problem by ignoring heterogeneity and assuming Wi-Fi environment with constant bandwidth (i.e., $C_i = C_j$, $\mathsf{B}_i = \mathsf{B}_j$ and B_i is a Dirac delta function). We setup pipelines of two lengths (4 and 8), and split the CNN based on Section IV-A. The odd and even indices correspond to computation and communication stages. We observe: 1) communication of the early stages (i.e., early

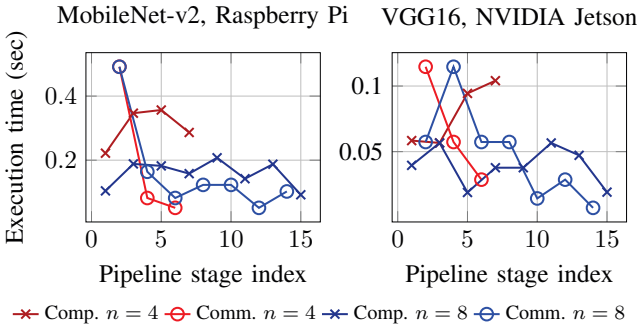

Fig. 1: Computation and communication time per stage

CNN layers) may bottleneck the overall pipeline throughput; 2) communication time of later stages decrease very sharply. Observation 1 motivates data compression (Section IV-C). Observation 2 implies that later stages are unlikely to become the performance bottleneck, even when we increase n or the variance of bandwidth B_i. Based on the observation, we simplify our design (Section IV-D) and analysis (Section V).

C. Load-Reduction for the Communication Stages

To improve throughput of the bottleneck communication stage, we compress the inter-device data by inserting suitable compressors into the pipeline. Compression of device i to $i+1$ communication should consist of the below steps:

- At device i: Compress layer-s_i output to a tensor $\boldsymbol{M}_i$.
- Between device i, $i+1$: Transfer compressed data $\boldsymbol{M}_i$.
- At device $i+1$: Decompress $\boldsymbol{M}_i$ to a tensor as close to the original layer-s_i activation as possible.

The compressor should satisfy the following requirements:

- Low overhead: Compression and decompression should be light-weight to maintain computation load-balance.
- High accuracy: Offline re-training should ensure high accuracy of the compressor-inserted CNN. The improved throughput after compression should lead to higher ν_{eff}.
- Flexibility: The compressor should be able to significantly reduce the data size and be easily configurable to various compression ratio when the bandwidth varies.

Define the compression ratio $\gamma_i \coloneqq \frac{\texttt{data}_{\mathbb{A}}(s_i)}{\texttt{size}(\boldsymbol{M}_i)}$, where $\texttt{size}\left(\cdot\right)$ returns the size of the tensor, and $\texttt{data}_{\mathbb{A}}\left(\cdot\right)$ returns the output activation size of the specified layer. We interpret $\boldsymbol{M}_i$ as the low-dimensional embedding of the original layer-s_i activation. Then the compression-decompression steps can be viewed as an encoding-decoding procedure. In other words, $\boldsymbol{M}_i$ is not necessarily similar to the original activation, as long as the decoder at device $i+1$ finds a good way to reconstruct the original activation. Such interpretation enables us to think beyond classic image compression algorithms such as JPEG [19], where visual features have to be preserved.

We propose to build our compressor upon the architecture of Auto Encoder (AE), a powerful deep learning model for data generation or reconstruction. The input tensor $\boldsymbol{X}_{\text{in}}$ goes through an encoding neural network and is transformed to

Algorithm 1 Pre-processing

Input: System specification $\mathcal{B} = \{\mathsf{B}_j \mid 1 \le j \le n-1\}$ and $\mathcal{C} = \{C_i \mid 1 \le i \le n\}$; CNN architecture $\mathbb{A}$

Output: Split point indices $\mathcal{S} = \{s_i \mid 1 \le i \le n-1\}$; Set of compressed CNNs and their accuracy $\mathcal{A}$, $\mathcal{N}$

1: ▷ Find split points
2: $\mathcal{W} \leftarrow \{W_i = \mathtt{ops}_{\mathbb{A}}(i) \mid 1 \le i \le m\}$
3: $s_0 \leftarrow 0$; $s_n \leftarrow m$;
4: Identify $\mathcal{S} = \{s_j\}$ by solving Equation 2
5: ▷ Determine compression ratio at each split point
6: $\mathcal{D} \leftarrow \{D_j = \mathtt{data}_{\mathbb{A}}(s_j) \mid 1 \le j \le n-1\}$
7: **for** $j = 1$ to $n-1$ **do**
8: $r_j \leftarrow \eta \cdot \frac{D_j}{\mathbb{E}[B_j]} T_{\mathrm{p}}$ ▷ Bound on compression ratio
9: $\mathcal{R}_j \leftarrow$ Set of potential compression ratio less than r_j
10: $\mathcal{R}_{\mathrm{CNN}} \leftarrow \{\mathcal{R}_j \mid 1 \le j \le n-1\}$
11: $\mathcal{A}_{\mathrm{cpr}} \leftarrow \left\{\mathbb{A}_{\gamma\times} \,\middle|\, \gamma \in \bigcup_j \mathcal{R}_j\right\}$
12: $\mathcal{A} \leftarrow \left\{\mathtt{fuse}\left(\mathbb{A}, \mathbb{A}_{\gamma_1\times}, \ldots, \mathbb{A}_{\gamma_{n-1}\times}\right) \mid \gamma_j \in \mathcal{R}_j\right\}$
13: **for** $\mathbb{A}_k \in \mathcal{A}$ **do**
14: Train $\mathbb{A}_k$ to obtain accuracy ν_k; Add ν_k to a set $\mathcal{N}$
15: **return** $\mathcal{S}$, $\mathcal{A}$, $\mathcal{N}$

the embedding tensor $\boldsymbol{X}'$. The embedding goes through a decoding neural network to become $\boldsymbol{X}_{\mathrm{out}}$. By proper architecture design and training configuration, $\boldsymbol{X}_{\mathrm{in}} \approx \boldsymbol{X}_{\mathrm{out}}$ and $\mathtt{size}(\boldsymbol{X}') \ll \mathtt{size}(\boldsymbol{X}_{\mathrm{in}}) = \mathtt{size}(\boldsymbol{X}_{\mathrm{out}})$. In our design, $\boldsymbol{X}_{\mathrm{in}}$ is the layer-s_i output activation at device i; $\boldsymbol{X}_{\mathrm{out}}$ is the layer-s_i+1 input activation at device $i+1$; $\boldsymbol{X}' = \boldsymbol{M}_i$.

There are various advantages to compress data using AE. First, the compressor can be seamlessly integrated into the original CNN, since they both are neural networks built upon convolutional layers. Secondly, the expressive power of multi-layer structure [20] of AE helps accuracy recovery. Lastly, by setting the layer parameters such as number of channels and stride, we can easily achieve arbitrary compression ratio.

a) Design of encoder/decoder architecture: We build encoder/decoder by stacking convolutional layers. To reduce computation overhead, we use the "inverted-residual" layers as the building block. For encoder, an inverted-residual layer contains depthwise separable convolution. For decoder, an inverted-residual layer contains depthwise separable *transposed* convolution. A stride larger than 1 shrinks the spatial dimension in the encoder, while expands the spatial dimension in the decoder. Denote $\mathbb{A}_{\gamma\times}$ as the AE with compression ratio γ. Table I summarizes $\mathbb{A}_{\gamma\times}$ used in our experiments.

b) Computation overhead: Observe from Table I that, even for compression ratio as high as 8, two layers in the encoder or decoder are sufficient. Since the original CNN contains tens to hundreds of convolutional layers, the overhead to compute $\mathbb{A}_{\gamma\times}$ (see Section VI-A) is negligible.

c) Training: Let $\mathbb{A}'$ be the CNN after inserting $\mathbb{A}_{\gamma\times}$ into $\mathbb{A}$ (Denote as: $\mathbb{A}' = \mathtt{fuse}(\mathbb{A}, \mathbb{A}_{\gamma\times})$). We perform end-to-end training of $\mathbb{A}'$, without factoring out the reconstruction step of $\mathbb{A}_{\gamma\times}$. The loss function is the cross entropy loss, measuring the difference between the ground-truth and the labels predicted by $\mathbb{A}'$. Such training allows fine-tuning of the weights in the original layers of $\mathbb{A}$ to compensate the reconstruction noise.

Summary of pre-processing The two main optimizations, splitting and compression are both offline before the pipeline deployment (Algorithm 1). Note from lines 8 to 12 that, at each split point, we identify a set of potential compression ratio and the corresponding compressors. In the next section, we show how to adaptively select the appropriate compression ratio in runtime, based on the real-time bandwidth measurement.

D. Adaptive Communication Compression

The insertion of compressor $\mathbb{A}_{\gamma\times}$ into the original $\mathbb{A}$ addresses the issue of low bandwidth. Further performance improvement can be achieved by considering the large variation of the bandwidth. Recall that we model B_i as a random variable following distribution B_i. If within the time window, we measure a low value of B_i, then the effective accuracy ν_{eff} may decrease due to increased number of frames being dropped. The best strategy then would be to insert a compressor with higher γ. On the other hand, if we measure a high value of B_i, then bandwidth at the split point i may not be the bottleneck. So we may replace the compressor with a lower γ so that ν_{eff} improves due to higher pre-trained accuracy ν.

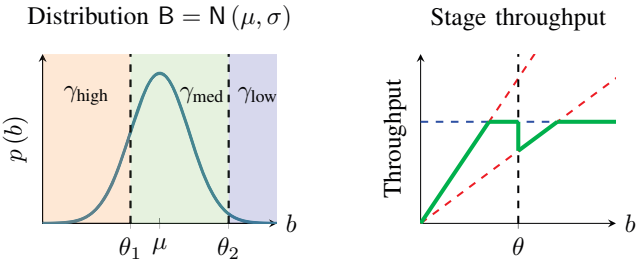

Fig. 2: Using various compressors based on threshold θ

The above intuition motivates us to develop an adaptive communication compression strategy. Algorithm 2 outlines the real-time inference procedure. At each split point, we calculate a set of threshold parameters Θ_i. The thresholds "divide" the bandwidth into multiple regions, each region associated with a compression ratio γ. In Figure 2, the two thresholds $\Theta_i = \{\theta_1, \theta_2\}$ divide the bandwidth into three regions. If the measurement b falls between θ_1 and θ_2, then at the split point, we use a compressor with γ_{med}. If b falls in other regions, other γ should be used. In our problem definition, the bandwidth changes slowly. So the reload of $\mathbb{A}'$ happens infrequently.

We make the following simplification to compute θ:

1) Only one communication stage can become bottleneck.
2) The input generation rate is larger than the throughput of the bottleneck computation stage (i.e., $T_{\mathrm{gen}} \ge T_{\mathrm{p}}$).

Point 1 is reasonable by the observation of Section IV-B. Point 2 is valid since T_{p} is known before inference.

Let γ_{left}, γ_{right} be the compression ratio at the two sides of θ. Let $\mathbb{A}_{\mathrm{left}}$, $\mathbb{A}_{\mathrm{right}}$, ν_{left}, ν_{right} be the CNN architecture and pre-trained accuracy ($\gamma_{\mathrm{left}} > \gamma_{\mathrm{right}}$, $\nu_{\mathrm{left}} < \nu_{\mathrm{right}}$). For discussion, we further assume $\gamma_{\mathrm{left}} = 2\gamma_{\mathrm{right}}$ and $0.5 \cdot \nu_{\mathrm{right}} < \nu_{\mathrm{left}} < \nu_{\mathrm{right}}$.

TABLE I: Architecture of $\mathbb{A}_{\gamma\times}$. Note, 1) Number of output channels depends on the number of channels in $\boldsymbol{X}_{\text{in}}$. E.g., $\boldsymbol{X}_{\text{in}}$ has 24 channels, the encoder with output channels of "$2\times \rightarrow 2\times$" contains two layers, both of 48 output channels; 2) Stride-2 shrinks (expands) spatial dimension in the encoder (decoder); 3) "Expansion ratio" is defined in Section II-A).

	Encoder				Decoder			
	Kernel size	Output channels	Expansion ratio	Stride	Kernel size	Output channels	Expansion ratio	Stride
$\gamma = 2$	3	$2\times$	1	2	3	$1\times$	0.5	2
$\gamma = 4$	$3 \rightarrow 3$	$1\times \rightarrow 1\times$	$1 \rightarrow 1$	$1 \rightarrow 2$	$3 \rightarrow 3$	$1\times \rightarrow 1\times$	$1 \rightarrow 1$	$2 \rightarrow 1$
$\gamma = 8$	$3 \rightarrow 3$	$2\times \rightarrow 2\times$	$1 \rightarrow 1$	$2 \rightarrow 2$	$3 \rightarrow 3$	$1\times \rightarrow 1\times$	$0.5 \rightarrow 1$	$2 \rightarrow 2$

Algorithm 2 Real-time inference

Input: Split points $\mathcal{S}$; Set of compressed CNNs $\mathcal{A}$; Bandwidth threshold parameters $\{\Theta_i \mid 1 \leq i \leq n-1\}$
Output: Inference pipeline executed on edge devices

```
for every reload time period do
    for each edge device i = 1 to n do                ▷ in parallel
        Measure current bandwidth b_i
        Determine A_{γ_i×} based on b_i and thresholds Θ_i
    A' ← Desired compressed CNN
    for each edge device i = 1 to n do                ▷ in parallel
        Load layers of A' based on s_{i-1}, s_i
    Continue pipelined inference
```

The optimal θ^* should be such that effective accuracy of $\mathbb{A}_{\text{left}}$ at $b = \theta^*$ equals effective accuracy of $\mathbb{A}_{\text{right}}$ at $b = \theta^*$. Now, to compute ν_{eff}, we visualize in Figure 2.B the change of pipeline stage throughput with respect to bandwidth b, under various scenarios. The horizontal dashed line (A) is the throughput bound T_{p} of computation stages. The two red dashed lines (B,C) represent the communication throughput under two compression ratio. Due to the assumption on the relative values of $\gamma_{\cdot}$ and $\nu_{\cdot}$ presented above, θ^* must fall between the intersection of A,B and the intersection of B,C:

$$T_{\text{p}} \cdot \nu_{\text{left}} = \frac{\gamma_{\text{right}} \cdot \theta^*}{D} \cdot \nu_{\text{right}} \tag{3}$$

where D is the amount of data traffic at the communication stage. The solid line in Figure 2.B shows the throughput of the overall pipeline with respect to b. At the transition point θ^*, the dropped throughput due to a less compressed CNN is compensated by the increased pre-trained accuracy.

V. Performance Analysis

The pipeline can be bottlenecked by either computation or communication. We derive the expectation of inference throughput for each case. $p_i(\cdot)$ and $P_i(\cdot)$ denote probability density function and cumulative distribution function of B_i.

a) Computation bottleneck: Let throughput of the slowest computation stage be T_{p}. According to our setup, T_{p} is fixed since $C_i \in \mathcal{C}$ are fixed. Let $\mathbb{P}_1$ be the probability of T_{p} becoming the bottleneck (i.e., all communication stages have larger throughput than T_{p}). Hence, $\mathbb{P}_1$ can be calculated by

$$\mathbb{P}_1 = \prod_i \mathbb{P}\left(\frac{B_i}{D_i} > T_{\text{p}}\right) = \prod_i \left(1 - P_i\left(D_i \cdot T_{\text{p}}\right)\right) \tag{4}$$

b) Communication bottleneck: The communication stage j with throughput τ becomes the bottleneck if and only if: 1) $T_{\text{p}} > \tau$; 2) Stage j has the smallest throughput among all communication stages. The probability of stage j becoming the bottleneck is $\prod_{i\neq j} \mathbb{P}\left(\frac{B_i}{D_i} > \tau\right) \cdot \left(\mathbb{P}\left(\frac{B_j}{D_j} = \tau\right)\right)$. The expected throughput in such case is

$$\mathbb{E}(T, j) = \int_{-\infty}^{T_{\text{p}}} \prod_{i\neq j} \left(1 - P_i(D_i \cdot \tau)\right) \cdot p_j\left(D_j \cdot \tau\right) D_j \cdot \tau d\tau \tag{5}$$

c) Expected throughput: The expected inference throughput can be calculated by combining the cases above:

$$\mathbb{E}[T] = \mathbb{P}_1 \cdot T_{\text{p}} + \sum_j \mathbb{E}(T, j) \tag{6}$$

d) Case study: According to the empirical measurement in [21], [22], B_i approximately follows Gaussian distribution. In the following, we analyze a system with 2 split points ($n = 3$), where B_1 and B_2 are i.i.d. random variables and $B_i \sim \mathsf{N}(\mu, \sigma)$ for $i = 1, 2$. The more general case with any n can be analyzed by following the same roadmap. By Equations 4, 5, 6, we have the expected throughput of the pipeline:

$$\begin{aligned}\mathbb{E}(T) &= \left(1 - \Phi(a_1)\right)\left(1 - \Phi(a_2)\right) T_{\text{p}} \\ &+ \frac{D_1}{\sigma} \int_{-\infty}^{T_{\text{p}}} \left(1 - \Phi\left(\frac{D_2\tau - \mu}{\sigma}\right)\right) \phi\left(\frac{D_1\tau - \mu}{\sigma}\right) \tau d\tau \\ &+ \frac{D_2}{\sigma} \int_{-\infty}^{T_{\text{p}}} \left(1 - \Phi\left(\frac{D_1\tau - \mu}{\sigma}\right)\right) \phi\left(\frac{D_2\tau - \mu}{\sigma}\right) \tau d\tau\end{aligned} \tag{7}$$

where $a_i = \frac{D_i T_{\text{p}} - \mu}{\sigma}$, $i = 1, 2$. We can derive the closed-form expression of the integral in Equation 7 with the help of bivariate normal cumulative (i.e., $BvN[\cdot]$).

e) Expected accuracy: By definition of expected accuracy, $\nu_{\text{eff}} = \frac{T \cdot \nu}{T_{\text{gen}}}$, we have $\mathbb{E}[\nu_{\text{eff}}] \propto \mathbb{E}[T \cdot \nu]$.

f) Analysis with adaptive compression: $\mathbb{E}[T]$ and $\mathbb{E}[\nu_{\text{eff}}]$ are functions of D_i, and thus functions of γ_i. We compute effective throughput and accuracy with minor modification of Equation 7. Now that the bandwidth probability distribution is "partitioned" by θ, the integration above becomes piece-wise.

VI. Experiments

We evaluate using MobileNet-v2 and two image classification datasets: CIFAR10, CIFAR100. We build the pipeline of Raspberry Pi 3B+. MobileNet-v2 consists of 19 convolutional

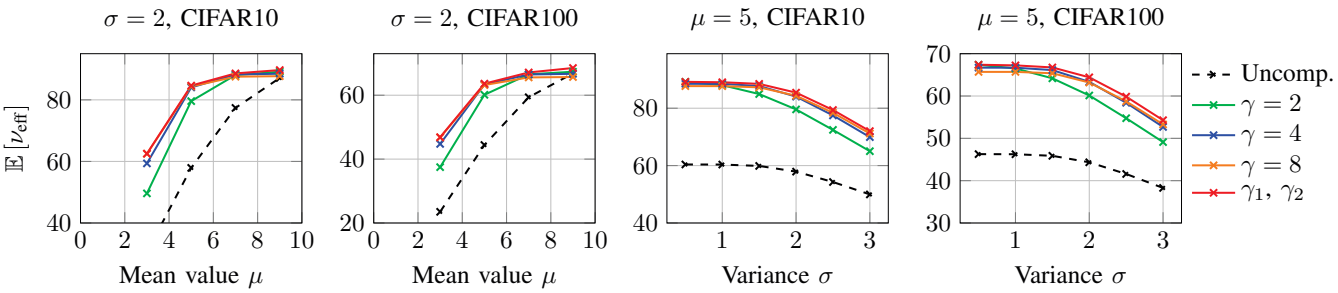

Fig. 3: Expected effective accuracy under various value of μ and σ, where $\mathsf{B}_i = \mathsf{N}(\mu, \sigma)$

TABLE II: MobileNet-v2: Accuracy & computation overhead

	CIFAR-10		CIFAR-100	
	Accuracy	Overhead	Accuracy	Overhead
Original	90.2	0.00×	69.1	0.00×
$\gamma = 2$	89.2	0.02×	67.4	0.02×
$\gamma = 4$	88.5	0.02×	66.7	0.02×
$\gamma = 8$	87.7	0.02×	65.7	0.02×

layers, where layer 1 and 19 are regular 3×3 convolutional layers, and layers 2 to 18 are inverted-residual blocks constructed by depthwise separable convolution. Both CIFAR10 and CIFAR100 consist of 32×32 R.G.B color images. Raspberry Pi contains a Broadcom BCM2837B0 quad-core A53 (ARMv8) CPU @1.4GHz, and a Broadcom Videocore-IV GPU. The RAM is 1GB LPDDR2. Raspberry Pi supports 2.4GHz and 5GHz 802.11b/g/n/ac Wi-Fi. We insert a 32GB Micro-SD card for storage. The inference pipeline consists of four Raspberry Pi devices. Devices in the pipeline communicate via Wi-Fi. We implement our code using Python3.7 and Tensorflow-Lite.

A. Evaluation on Compressors

Based on our splitting algorithm (Section IV-A), the splitting points are $\mathcal{S} = \{4, 11, 15\}$. The communication bottleneck only happens at the first splitting point. We thus insert compressors of $\gamma = 2, 4, 8$ between device 1 and 2 of the pipeline. The compressor architectures are defined by Table I.

Table II summarize the pre-trained accuracy (i.e., ν) and the computation overhead due to the additional compressor layers. Clearly, 1) the proposed end-to-end training ensures high accuracy even when the compression ratio is very high; 2) the additional computation load due to the inserted compressor is at most 2% of the computation load of the original model. Therefore, after inserting the compressor, there is no need to re-split layers for re-balance of computation load.

B. Evaluation on End-to-End Performance

We evaluate the expected effective accuracy $\mathbb{E}[\nu_{\text{eff}}]$ under various network conditions. Figure 3 is measured by 1) fixing the variance to be $\sigma = 2$Mbps, and changing the mean value from $\mu = 3$Mbps to 9Mbps; 2) fixing the mean to be $\sigma = 5$Mbps, and changing the variance from $\sigma = 0.5$Mbps to 3Mbps. In all experiment, we set the input image generation rate to be the throughput of the bottleneck computation stage (i.e., $T_{\text{gen}} = T_{\text{p}}$). So $\nu_{\text{eff}} = \nu$ if and only if the communication stages never become the system bottleneck.

Effectiveness of compression Data compression significantly improves the overall effective accuracy. Effective accuracy of the uncompressed model catches up with that of the compressed model, only for very large μ. For the two plots of MobileNet-v2 in Figure 3, when $\mu = 9$, we have $\frac{\mu}{D_1} \approx 1.7 \times T_{\text{p}}$. In addition, we observe that higher compression ratio is more useful when the network condition is bad. For Figure 3, larger γ leads to significant accuracy improvement compared with smaller γ, when μ is small. Effective accuracy of CNNs with small γ eventually becomes better when μ keeps increasing. This is because when μ is very large, the communication stage is very unlikely to become the bottleneck, and ν then becomes the dominant term in the calculation of ν_{eff}. Lastly, larger compression ratio are more beneficial when the variance of the bandwidth is larger.

Effectiveness of adaptive compression With adaptive compression (red line labeled "γ_1, γ_2"), we use a single θ to support two compression ratio between devices 1 and 2. Inference with adaptive data compression almost always achieve the best performance, regardless of the bandwidth condition. For example, for MobileNet-v2 on CIFAR100, the adaptive compression scheme clearly leads to higher accuracy than other single compression ratio schemes. Under large bandwidth variance, we can hardly identify a single compression ratio suitable for all scenarios. If we allow the communication stage to choose among more compressors, we expect higher accuracy than the current simple adaptive scheme.

VII. Conclusion

We have proposed a framework for improving throughput and accuracy of pipelined CNN inference on the edge. To improve the inter-device communication performance, we have proposed an AE-based adaptive compression scheme to 1) significantly reduce the activation size with negligible accuracy loss, and 2) optimizes expected accuracy by adapting the compression ratio given the current bandwidth.

We will extend our adaptive compression scheme to support multiple communication stages. We will apply data quantization for higher compression ratio. We will evaluate on more CNNs and devices to better understand the performance gain.

REFERENCES

[1] X. Xie and K.-H. Kim, "Source compression with bounded dnn perception loss for iot edge computer vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3300061.3345448

[2] 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," *arXiv preprint arXiv:1602.07360*, 2016.

[3] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[6] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," *arXiv preprint arXiv:1905.02244*, 2019.

[7] E. Li, Z. Zhou, and X. Chen, "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy," in *Proceedings of the 2018 Workshop on Mobile Edge Communications*. ACM, 2018, pp. 31–36.

[8] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*. ACM, 2019, pp. 21–26.

[9] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[10] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[14] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 82–95.

[15] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[17] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 389–400.

[18] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[19] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[20] R. Eldan and O. Shamir, "The power of depth for feedforward neural networks," in *Conference on learning theory*, 2016, pp. 907–940.

[21] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone, "Network offloading policies for cloud robotics: a learning-based approach," *arXiv preprint arXiv:1902.05703*, 2019.

[22] I. Cardei, A. Agarwal, B. Alhalabi, T. Tavtilov, T. Khoshgoftaar, and P.-P. Beaujean, "Software and communications architecture for prognosis and health monitoring of ocean-based power generator," in *2011 IEEE International Systems Conference*. IEEE, 2011, pp. 353–360.