AMVP: Adaptive CNN-based Multitask Video Processing on Mobile Stream Processing Platforms

Conference Paper · November 2020

DOI: 10.1109/SEC50012.2020.00015

Chao Mengyuan

Texas A&M University

Radu Stoleru

Texas A&M University

Liuyi Jin

Texas A&M University

AMVP: **A**daptive CNN-based **M**ultitask **V**ideo **P**rocessing on Mobile Stream Processing Platforms

Mengyuan Chao, Radu Stoleru, Liuyi Jin, Shuochao Yao*, Maxwell Maurice†, Roger Blalock†
Department of Computer Science and Engineering, Texas A&M University
*Department of Computer Science, George Mason University
†National Institute of Standards and Technology (NIST)
{*chaomengyuan, stoleru, liuyi*}@*tamu.edu*, *shuochao*@*gmu.edu*, {*maxwell.maurice, roger.blalock*}@*nist.gov*

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00015

***Abstract*—The popularity of video cameras has spawned a new type of application called multitask video processing, which uses multiple CNNs to obtain different information of interests from a raw video stream. Unfortunately, the huge resource requirements of CNNs make the concurrent execution of multiple CNNs on a single resource-constrained mobile device challenging. Existing solutions solve this challenge by offloading CNN models to the cloud or edge server, compressing CNN models to fit the mobile device, or sharing some common parts of multiple CNN models. Most of these solutions, however, use the above offloading, compression or sharing strategies in a separate manner, which fail to adapt to the complex edge computing scenario well.**

In this paper, to solve the above limitation, we propose *AMVP*, an adaptive execution framework for CNN-based multitask video processing, which elegantly integrates the strategies of CNN layer sharing, feature compression, and model offloading. First, *AMVP* reduces the total computation workload of multiple CNN inference by sharing some common frozen CNN layers. Second, *AMVP* supports distributed CNN inference by splitting big CNNs into smaller components running on different devices. Third, *AMVP* leverages a quantization-based feature compression mechanism to reduce the feature transmission traffic size between two separate CNN components. We conduct extensive experiments on *AMVP* and the experimental results show that our *AMVP* framework can adapt to different performance goals and execution environments. Compared to two baseline approaches that only share or offload CNN layers, *AMVP* achieves up to 61% lower latency and 10% higher throughput with comparative accuracy.

I. Introduction

With the wide deployment of video cameras, *multitask video processing* is having an increasingly important impact on our daily life. For example, policemen spot stolen vehicles, locate traffic accidents and track criminals by analyzing videos taken by the road cameras, office workers get notified of the burglary, theft and fallen seniors at home by processing videos taken by the home cameras, disaster responders search for victims and hazard symbols by processing videos streamed from the UAV cameras, etc. These applications have a common characteristic: *they process a raw video stream with multiple vision analysis pipelines to acquire different information of interests.*

Nowadays, the core technology to enable complicated vision applications is Convolutional Neural Networks (CNNs), which have replaced traditional computer vision methods to become the mainstream technology for vision processing due to their distinguished accuracy and performance [1]–[4]. In this case, *CNN-based multitask video processing*, which utilizes multiple sophisticated CNNs to run vision analysis tasks on a given raw video stream becomes popular [5]–[7]. However, running multiple computational intensive CNNs on a resource constrained device (e.g., a surveillance camera) to achieve the user desired application performance (e.g., high accuracy and low latency) is not easy. Additionally, the fact that different users may have different performance preference on different tasks makes this problem more complicated and challenging.

Existing solutions for addressing the above challenge can be roughly divided into three categories: offloading, compression and sharing. First, many offloading strategies [8]–[12] assume that there is a connection between mobile devices and remote cloud (or a nearby edge server) so that parts of CNNs can be offloaded there to achieve the desired performance. However, a stable connection to the remote cloud is not always available for specific scenarios like disaster response [13] or military operations and deploying a powerful edge server near each video camera is costly. The other offloading strategies [14]–[18] distribute a whole CNN model to run on several wireless connected IoT devices. However, they only consider the single CNN scenario, which is simpler than the multi-CNNs case we consider. Second, the model compression strategies construct efficient CNN models for mobile devices through different compression techniques, such as low-rank expansions [19], parameter quantization [20], pruning and Huffman coding [21], fully factorized convolution [22], depth-wise separable convolution [23], and channel-wise sparse connection [24], etc. Unfortunately, these solutions provide a one-for-all model, i.e., a fixed model compression technique is utilized for different performance goals. A recent work [25] employs on-demand compression, which applies proper compression techniques on different CNN layers to achieve an optimal balance between performance goals and resource constraints. However, it still focuses on the single CNN case. Third, the sharing strategies reduce the computation costs [9] and memory footprint [26] by sharing common layers among multiple CNNs. However, the existing solutions run multiple CNNs on a single device, so the performance improvement is restricted by the device resource and the scalability is poor.

Different from existing works which use the above strategies separately, in this paper, we combine these strategies together

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00015

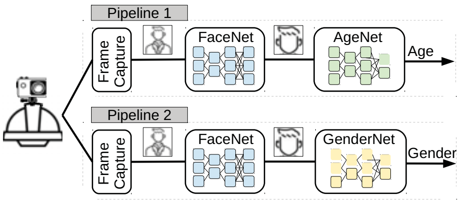


Fig. 1: An example of CNN-based multitask video processing, which consists of two vision analysis pipelines and four CNNs.

as *AMVP*, an Adaptive execution framework for CNN-based Multitask Video Processing on wireless connected mobile/IoT devices. First, *AMVP* reduces the computation cost by sharing some common components among different analysis pipelines. For the distinct components which run different vision tasks, *AMVP* provides multiple CNN candidates for each of them. These candidates are pre-trained from some well-known base CNNs (e.g., mobileNet [27], resNet [28]) via transfer learning. If two distinct application components select CNNs derived from the same base CNN, they can share some common frozen layers to reduce the total computation cost. There is a trade-off between the number of common frozen layers different CNNs can share and the inference accuracy each CNN can achieve. *AMVP* aims to keep an optimal balance between them based on the user performance goals and available computing resources. Second, *AMVP* supports distributed CNN inference by splitting big CNNs into smaller components running on different devices. In order to reduce the communication costs, *AMVP* leverages a quantization-based feature compression methodd to compress the feature maps transmitted between two separate CNN components. Third, *AMVP* is built on top of a mobile stream processing platform called MStorm [29], where it maintains an adaptive scheduler to choose appropriate CNN candidates for different pipeline components and assign these components to appropriate devices to achieve the user desired performance goals. We implement a prototype system of *AMVP* on Android phones and demonstrate its superiority by running a CNN-based multitask video analysis application. The experimental results show that, compared to two baseline solutions, *AMVP* achieves up to 61% lower latency and 10% higher throughput with comparative accuracy.

In summary, this paper makes the following contributions:

- It shows the possibility of combing three types of orthogonal strategies (i.e. offloading, compression and sharing) to support the execution of multiple computational intensive CNNs on resource-constrained mobile/IoT devices.
- It designs an adaptive framework which chooses the most appropriate CNNs implementation and running device for each pipeline component based on the user performance goals and the actually available system resources .
- It implements a prototype system on Android phones to prove its superiority over two status quo approaches in supporting CNN-based multitask video processing.

The rest of this paper is organized as follows. In Section II, we introduce background and motivation. Then, in Section III, we describe the system overview and give more design details in Section IV. Following that, we describe the system implementation and show some experimental results in Section V. In Section VI, we introduce some related works and we conclude this paper and propose some future work in Section VII.

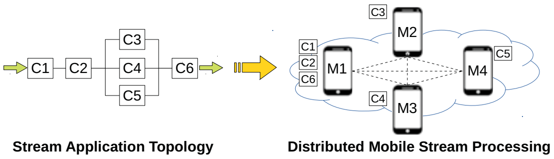

C1
C2
C3
C4
C5
C6
Stream Application Topology
C3
M2
C1
C2
C6
M1
C5
M4
C4
M3
Distributed Mobile Stream Processing

Fig. 2: An example of Distributed Mobile Stream Processing (DMSP), which assigns six components of a stream application to execute on four wireless-connected mobile devices.

II. Background and Motivation

A. CNN-based Multitask Video Processing

Multitask video processing is a new category of applications which applies multiple different analysis pipelines on a given raw video stream to run various vision processing tasks, such as object detection, people re-identification, image classification, activity recognition, etc. CNNs, because of distinguished accuracy and effectiveness, have replaced traditional computer vision methods to be the mainstream technology to implement complex vision tasks. In this case, CNN-based multitask video processing becomes increasingly popular in our lives. Fig. 1 is an example of CNN-based multitask video processing which consists of two vision analysis pipelines and four CNNs. It identifies both age and gender of survivors shown up in a video stream from a helmet camera, which helps disaster responders to collect the basic information about survivors during recuse.

To deal with issues such as intermittent connectivity, bandwidth limitation, real-time requirements, privacy concern, etc., it is common to run CNN-based multitask video processing at the edge rather than in the cloud. However, executing multiple CNNs concurrently is extremely computational intensive while edge devices usually have very limited computing resources. In addition, these resources are shared with other applications, which makes the actual resources available to run CNNs even fewer. Moreover, the users expect to achieve good application performance and the performance goals may change from time to time. All these factors make running CNN-based multitask video processing at the edge challenging.

B. Distributed Mobile Stream Processing

Distributed Mobile Stream Processing (DMSP) is an emerging computing paradigm [30]–[33] that supports online stream processing at the edge. Different from previous systems which offload computation tasks to a nearby edge server, DMSP uses resources of nearby mobile devices to conduct real-time stream processing. In DMSP, each stream application is represented as a graph called topology. Each topology consists of several logical units called component. Each component implements

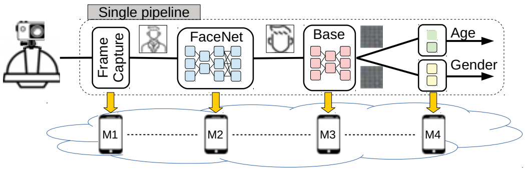


Fig. 3: Motivation of *AMVP*: sharing components and layers of multiple CNNs and running them on multiple mobile devices to improve the performance of multitask video processing.

an independent functionality of a whole pipeline. To execute a stream application on a DMSP platform, the user at first needs to submit an application topology. Then, the DMSP platform assigns different components to proper devices to perform distributed stream processing. Fig. 2 shows an example of DMSP, where six components C1-C6 of the stream application are assigned to run on four wireless-connected mobile devices M1-M4 to perform distributed stream processing.

Although DMSP makes it easier to deploy and scale stream processing at the edge, there are still some challenges to solve. First, in DMSP, mobile devices are not dedicated to do stream processing, other applications need to execute on them as well. This makes the actual available resources for DMSP uncertain. It would be great if a component of a DMSP application could adjust its workload based on the free resources of the device. Second, mobile devices in DMSP are connected by wireless networks that have dynamic bandwidths. It would be great if the data traffic size between two adjacent components can be reduced by some compression method and adjusted according to the dynamic networks.

C. Motivation and Challenges

In this paper, our goal is to develop an adaptive execution framework for the CNN-based multitask video processing on a DMSP platform. Our motivation comes from two aspects: On the one hand, a DMSP platform can provide more computation resources to CNN-based multitask video processing to achieve better performance. On the other hand, CNNs offer an elastic trade-off between the inference accuracy and computation cost, which perfectly match the aforementioned adaptive computing and communication requirements of DMSP. Thereby, as shown in Fig. 3, running CNN-based multitask video processing on a DMSP platform is a perfect win-win choice.

However, to enable the above combination, there are several challenges to overcome. First, we need to combine different pipelines in Fig. 1 into a single pipeline by sharing common components. Second, we need to find an approach to enable different CNNs to share different common layers and adjust total computation workload. Third, we need to implement an adaptive scheduler which chooses the most appropriate CNN candidates for each application component. Finally, we need a compression method to reduce the feature transmission size between two separate CNN parts running on two devices. In the following, we overcome these challenges on by one.

III. *AMVP* Architecture

Fig. 4 illustrates the architecture of *AMVP*, which consists of four stages: model training, model splitting, model profiling and model selection & task assignment. The first three stages are performed offline while the last state is performed online.

Model training. *AMVP* trains different CNNs for different vision processing tasks through transfer learning. It first takes a well-known pre-trained model, such as mobileNet or resNet, as vanilla model. Then, it replaces the classifier layers of the vanilla model and freezes some of its base layers. Next, *AMVP* trains the model with an input dataset (e.g., emotion dataset) and outputs a new CNN (e.g., emotionNet) for a specific vision task (e.g., emotion analysis). By using different vanilla models, replacing with different classifier layers, freezing different base layers and training with different datasets, CNNs with different accuracy and computation workload are obtained for different vision processing tasks. This stage is performed offline on a computer server using TensorFlow and Keras framework, with different pre-trained models and datasets as input, a set of .h5 format CNN models as output.

Model splitting. Since big CNNs might be too computationally intensive for a resource-constrained mobile device, they require to be split into two parts and run on different mobile devices to achieve the desirable performance. To this end, after getting a group of .h5 CNN models from the model training stage, *AMVP* splits each .h5 model into two and converts them to .tflite format for mobile devices. To adapt to the uncertain available resources at mobile devices, *AMVP* splits each .h5 model at different splitting points, resulting in different .tflite model pairs. For each pair, the first part extracts intermediate features from the input and the second part makes inference on the features output by the first part. For different pairs, the computation costs of the first and second parts, as well as the features between them, are different. This stage is performed offline using Keras, with different .h5 CNN models as input and a set of .tflite model pairs for each .h5 model as output.

Model profiling. After getting a set of .tflite model pairs for each vision task, *AMVP* profiles each pair of .tflite models on mobile devices to obtain the execution latency, memory footprint, inference accuracy of each part and the traffic size of feature transmission between two separate parts. To get an accurate baseline, when profiling these parameters, mobile devices are in an idle state, i.e., except for some necessary system services, no other applications are running. This stage is conducted offline using TensorFlow benchmark tools, with .tflite models as input and diverse profile information as output.

Model selection & task assignment. Finally, after getting a group of .tflite models and corresponding profile information for each vision analysis task, *AMVP* requires to assign a CNN-based multitask video processing pipeline to run on a DMSP platform to achieve the desirable performance. To this end, *AMVP* uses a runtime resource and network monitor to obtain the actually available resources and network speed at each mobile device. It also uses an APP topology and user preference manager to get the APP topology and performance

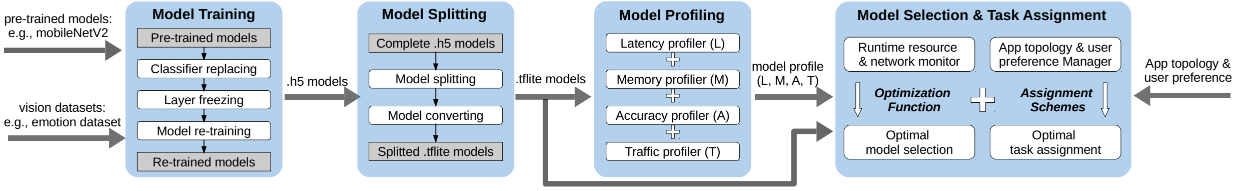


Fig. 4: AMVP architecture, which includes model training, splitting, profiling, selection and assignment.

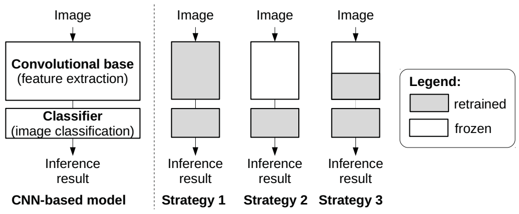


Fig. 5: Transfer learning strategies for CNN models

goals from the user. Then, *AMVP* chooses an optimal CNN candidate for each application component and assgins all the components to the optimal devices based on an optimization function and a task assignment scheme. A detailed description about optimization function and scheme is given in the next section. This stage is conducted online, with .tflite models and profile information, available resources and network condition, application topology and performance goals as input, with the optimal model selection and task assignment as output.

IV. Design of *AMVP*

A. *Transfer Learning and Layer Sharing*

In *AMVP*, all CNNs for different vision analytic tasks are obtained through transfer learning – an efficient approach of building CNNs for new tasks. Instead of learning from scratch, transfer learning starts with some patterns learned from solving similar problems. In computer vision, these patterns are pre-trained CNN models (e.g., mobileNet, resNet, etc.) trained on large benchmark datasets (e.g., ImageNet, CIFAR, etc.).

As shown in Fig. 5, a pre-trained CNN model can be divided into two parts: the convolutional base that consists of stacked convolutional and pooling layers and the classifier that consists of fully connected layers. The goal of a convolutional base is to extract features from an input image whereas the goal of a classifier is to classify the input image based on the extracted features. The convolutional base learns hierarchical features: the features learned by lower-layers are general whereas the features learned by higher-layers are specialized [34]. General features can be shared among different tasks whereas specific features strongly depend on the specific tasks and datasets. The key idea of *AMVP* is to reduce the total computation cost of an application by sharing common layers that generate general features among different CNNs while using the rest layers of each CNN to perform task-specific inference.

Based on transfer learning, *AMVP* trains CNNs for different video tasks of an application as follows. At first, it replaces the original classifier in a pre-trained model with a new classifier. Then, it tunes the parameters of the new model with strategies in Fig. 5. Strategy 1 retrains all the parameters of the model from scratch, which needs a large training dataset and a lot of computation. Strategy 2 freezes the whole convolutional base and only retrains the classifier. It uses a pre-trained model as a fixed feature extractor, which is suitable for training CNNs for new tasks similar to the original tasks with small datasets. Strategy 3 is a compromise between Strategy 1 and Strategy 2, which freezes some convolutional layers and retrains the rest. As a rule of thumb, when a new task has a small dataset, freezing more layers avoids overfitting; however, when a new task has a large dataset, retraining more layers improves the accuracy. With different frozen layers, *AMVP* generates a list of CNNs for each task with different accuracy (as shown in Fig. 9). We assume CNNs for different tasks in an application are generated from the same pre-trained model. In this case, *AMVP* can share some common frozen layers among multiple CNNs to reduce the overall computation and memory costs. According to our experience, there is a trade-off between the number of frozen layers different CNNs actually share and the inference accuracy each CNN achieves. Generally, the more layers different CNNs share, the more computation costs can be reduced, but the less specific each CNN will be and the lower inference accuracy each CNN can achieve. Therefore, one challenge of *AMVP* design is to choose appropriate frozen layers for each CNN to balance between the total computation cost and the inference accuracy of each task.

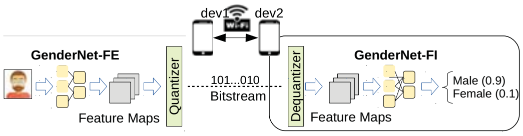


Fig. 6: Compression and decompression of feature maps

B. *Model Splitting and Feature Compression*

In *AMVP*, to enable distributed CNN execution, a big CNN model is split into two separate parts: Feature Extraction (FE) and Feature Inference (FI). FE takes a raw image as input and outputs features extracted from that image; FI takes features extracted from a raw image as input and outputs the inference results. There are multiple splitting points where a CNN model can be split, resulting in a group of (FE, FI) pairs. To improve the processing throughput, FE and FI of a pair are scheduled to run on different mobile devices. However, due to the large data size of intermediate features and dynamic bandwidth of wireless networks, how to efficiently transfer features from FE to FI becomes a big challenge. In *AMVP*, we use quantization-based compression method to reduce the data size of feature transmission. The whole processes is shown in Fig. 6, where the features output by a FE is quantized by a quantizer before sent to a FI. After FI receives the quantized features, it uses corresponding dequantizer to recover the original feature.

Splitting: There are multiple splitting points to divide a CNN into two separate parts. For different splitting points, the traffic size for feature transmission between two separate parts are different. Usually, the size of features gradually gets reduced along with the inference process [35]. The reason behind is: features at the lower layers represent lower level information, which are more detailed and fragmented; however, features at the higher layers represent higher level information converged from the lower level, which are more abstract and advanced. Usually, the lower level information is more than the higher level. However, this characteristic of CNNs does not mean that *AMVP* should always split a CNN at a higher layer to reduce the communication cost. In fact, different from the asymmetric camera-cloud architecture [36] where the computation cost at the cloud is negligible and the communication cost between a camera and the cloud constitutes the main part of the total latency, our architecture is more challenging because both the computation cost at two devices and the communication cost between them constitutes the main delay. Therefore, in *AMVP*, except for the communication latency, we also consider the computation balance of two separate parts. Although splitting at a higher layer saves communication costs, computation costs of two parts become less balanced, which is not beneficial for achieving high processing throughput and low latency.

Quantization: *AMVP* adopts quantiztion-based compression to reduce the communication cost of feature transmission [37]. The features output by FE is quantized to n-bit precision by a uniform quantizer defined as follows:

$$\overline{\mathbf{F}} = \lfloor \frac{\mathbf{F} - \min(\mathbf{F})}{\max(\mathbf{F}) - \min(\mathbf{F})} \cdot (2^n - 1) \rceil \tag{1}$$

where $\mathbf{F} \in \mathbb{R}^{H \times W \times C}$ is the tensor containing the feature map data with H as height, W as width, and C as channels. $\min(\mathbf{F})$ and $\max(\mathbf{F})$ denote the minimum and the maximum values in $\mathbf{F}$, respectively. $\overline{\mathbf{F}}$ denotes the quantized feature tensor and $\lfloor \cdot \rceil$ denotes a function rounding to the nearest integer. When the quantized feature tensor $\overline{\mathbf{F}}$, $\min(\mathbf{F})$ and $\max(\mathbf{F})$ are obtained at the FI of a CNN, $\overline{\mathbf{F}}$ is dequantized by a uniform dequantizer:

$$\widehat{\mathbf{F}} = \frac{\max(\mathbf{F}) - \min(\mathbf{F})}{2^n - 1} \cdot \overline{\mathbf{F}} + \min(\mathbf{F}) \tag{2}$$

where $\widehat{\mathbf{F}}$ denotes the dequantized feature tensor. It deserves to be mentioned that, although $\widehat{\mathbf{F}}$ is not exactly equal to $\mathbf{F}$, some research [38], [39] shows that, when the n value is above a threshold, the quantization process has a negligible effect on the accuracy of image classification and object detection. As described later in Section V-B, *AMVP* uses $n = 8$ to compress the feature most while keep the accuracy close to original.

Expect for quantiztion-based compression, other compression methods, such as GZIP, ZLIB, BZIP2, LEMA, JPEG2000, HEVC [40]–[42], can also be applied to further compress the features transmitted from FE to FI. We leave the interface for supporting more compression methods as the future work.

Compression metric: We leverage three metrics to evaluate the quantization-based feature compression method. The first metric is compression rate (CR), which is defined as:

$$CR = \frac{Feature\ size\ after\ compression}{Feature\ size\ before\ compression}. \tag{3}$$

The second metric is fidelity, which evaluates the information loss of dequantized features for image classification. It is calculated by comparing the original onehot classification results with the outputs inferred from the dequantized features [35]:

$$Fidelity = 1 - \frac{1}{2N} \sum_{i=1}^{N} Hamming(O_i^{og}, O_i^{cp}) \tag{4}$$

where O_i^{og} denotes the original onehot classification result of image sample i and O_i^{cp} denotes the onehot output inferred from the dequantized features. $Hamming(\cdot)$ is the hamming distance function and N denotes the total number of samples.

The third metric is compression benefits (CPB), which compares the total latency for transmitting features in compressed format and latency for transmitting features in original format. The concrete definition is as follows:

$$CPB = OTR - (QT + TR + DQ) \tag{5}$$

where OTR is the latency for transmitting features in original format, QT is the latency for feature quantization, TR is the latency for transmitting features in compressed format and DQ is the latency for feature dequantization.

C. Model Selection and Task Assignment

In *AMVP*, a set of .tflite models and profiles are deployed on mobile devices in advance, *AMVP* needs to choose an appropriate model for each vision task and assign the whole CNN-based multitask video processing pipeline to proper devices of a DMSP platform to achieve the desirable performance. We name this procedure as Model Selection and Task Assignment (MSTA) and formulate it mathematically as follows.

Let S denote the set of vision analytic tasks in a multitask video processing application and let A_s, L_s and T_s denote the user setting goals for inference accuracy, processing latency and processing throughput for task $s \in S$, respectively. Let M_s represent the set of all available CNN candidates for task s and let $m_s \in M_s$ denote a specific candidate with specific pre-trained CNN type and frozen layers. In addition, we use $\boldsymbol{m_s} = (m_s^1, m_s^2)$ to denote a specific separation of CNN model m_s, which consists of two separate parts m_s^1 and m_s^2. The size of features transmitted from m_s^1 to m_s^2 is denoted as $f_{m_s^1 m_s^2}$ and the communication bandwidth between devices k_1, k_2 that execute m_s^1, m_s^2 is denoted as $b_{k_1k_2}$. Then, the cost function for task s is defined as follows:

$$\begin{aligned} C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s) = \alpha_s \cdot & \frac{A_s - A(\boldsymbol{m_s})}{A_s} \\ + \beta_s \cdot & \frac{\max(0, L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) - L_s)}{L(\boldsymbol{m_s}, \boldsymbol{u_s^k})} \\ + \gamma_s \cdot & \frac{\max(0, T_s - T(\boldsymbol{m_s}, \boldsymbol{u_s^k}))}{T_s} \end{aligned} \tag{6}$$

where $A(\boldsymbol{m_s})$ is the inference accuracy of $\boldsymbol{m_s}$; $L(\boldsymbol{m_s}, \boldsymbol{u_s^k})$ is the processing latency defined as

$$L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) = \frac{l_{m_s^1}^{k_1}}{u_{m_s^1}^{k_1}} + \frac{l_{m_s^2}^{k_2}}{u_{m_s^2}^{k_2}} + D(f_{m_s^1m_s^2}, b_{k_1k_2}) \tag{7}$$

where $\boldsymbol{u_s^k} = (u_{m_s^1}^{k_1}, u_{m_s^2}^{k_2})$, $u_{m_s^i}^{k_j} \in (0, 1]$ denotes the percentage of computing resources allocated to model m_s^i at device k_j, $l_{m_s^i}^{k_j}$ denotes the processing latency of m_s^i when 100% computing resources of k_j are allocated to m_s^i, and $D(f_{m_s^1m_s^2}, b_{k_1k_2})$ denotes the latency for delivering the features from k_1 to k_2; $T(\boldsymbol{m_s}, \boldsymbol{u_s^k})$ is the processing throughput defined as

$$T(\boldsymbol{m_s}, \boldsymbol{u_s^k}) = \min\{u_{m_s^1}^{k_1} t_{m_s^1}^{k_1}, u_{m_s^2}^{k_2} t_{m_s^2}^{k_2}, \frac{1}{D(f_{m_s^1m_s^2}, b_{k_1k_2})}\} \tag{8}$$

where $t_{m_s^i}^{k_j}$ represents the processing throughput of m_s^i when 100% computing resources of k_j are allocated to m_s^i.

In the cost function, the first term promotes to select a CNN candidate with the highest inference accuracy, the second term penalizes choosing a CNN candidate that achieves a processing latency higher than the latency goal L_s and the third term penalizes choosing a CNN candidate that achieves a processing throughput lower than the throughput goal T_s. Since the video stream is streamed at a fixed frame rate, there is no reward to achieve a latency lower than L_s and a throughput higher than T_s. Therefore, in Eq. 6, we use a $\max$ function for the second and third terms. To allow the trade-off among accuracy, latency and throughput, parameters $\alpha_s, \beta_s, \gamma_s \in [0, 1]$, $\alpha_s + \beta_s + \gamma_s = 1$, can be set by the user to indicate the importance of accuracy, latency and throughput, respectively. The larger a parameter is, the more important the corresponding metric is.

It deserves to be mentioned that, asking different parameters from users is not intuitive. Therefore, in the future, instead of asking the users for the concrete budgets and preferences, we will provide several levels of services for the users to choose with the words they can understand. Inside the system, we can map the chosen service level to corresponding parameters.

Given the cost function of each task, *AMVP* applies a widely used MinMaxCost scheme [26] to perform the model selection and task assignment, which minimizes the cost of the task that has the largest cost. The optimization problem of this scheme is formulated as follows:

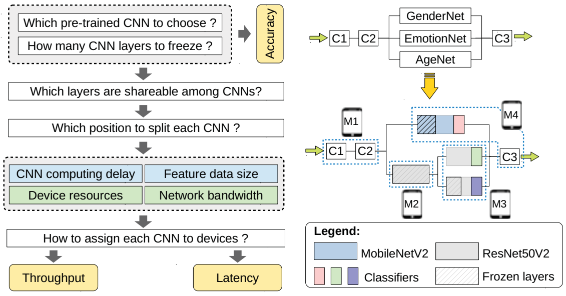
$$\underset{\boldsymbol{m_s}, \boldsymbol{u_s^k}}{\text{minimize}} \quad C \tag{9}$$

$$\text{subject to:} \quad \forall s : C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s) \leq C, \tag{10}$$

$$\forall k : \sum_{\{m_s^i\}} u_{m_s^i}^k \leq U_k, \tag{11}$$

$$\forall k : \sum_{\{m_s^i\}} r_{m_s^i}^k \leq R_k \tag{12}$$

where the cost of any task k must be smaller than C where C is minimized. $\{m_s^i\}$ is a CNN model set where all the models inside are different. At device k, $u_{m_s^i}^k$ denotes the percentage of computing resources allocated to model m_s^i, U_k denotes the percentage of total available computing resources, $r_{m_s^i}^k$ denotes the runtime memory usage of model m_s^i and R_k denotes the total available memory. With the MinMaxCost scheme, *AMVP* assigns resources to all the tasks of video processing in a fair way so that there is no obvious performance bottleneck.



(a) Basic workflow of MSTA

(b) An example of MSTA

Fig. 7: Model Selection and Task Assignment (MSTA)

In order to solve the above computationally hard nonlinear optimization problem, *AMVP* uses a greedy heuristic algorithm to get an approximate solution. The key idea of this algorithm is built on a basic workflow of MSTA shown in Fig. 7a. First, each task selects a specific CNN implementation from all the available candidates, which implies the following information: 1) the used pre-trained CNN model; 2) the frozen CNN layers. This selection directly determines the inference accuracy that each task can achieve. Then, after each task chooses its own CNN implementation, information about how many common frozen layers are shareable among different CNNs are easy to obtain. To minimize the total computation workload, different CNNs try to share as many common frozen layers as possible. With this shared layer information, each related CNN can be split into two sub-parts. Since the profile of each CNN sub-part is acquired in advance, *AMVP* finally assigns all the sub-parts to proper devices based on the monitored system status. Based on the assignment, throughput and latency of each task is determined. A detailed description of this greedy algorithm is in Algorithm 1. First, we use $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ to represent the final model selection and task assignment and we set the minimum maximum cost value $minMaxCost$ as 1 (line 1-2). Then, we

Algorithm 1: MSTA(), a greedy algorithm

```
Input  : ∀ task $s$: $M_s$, $A_s$, $L_s$, $T_s$, $\alpha_s$, $\beta_s$, $\gamma_s$;
         ∀ device $k, k'$: $U_k$, $R_k$, $b_{kk'}$;
         ∀ model $m_s^i$ and ∀ device $k$: $l_{m_s^i}^k, t_{m_s^i}^k$
Output: Model selection $\overline{\boldsymbol{Q}}$ and task assignment $\overline{\boldsymbol{X}}$
1  $\overline{\boldsymbol{Q}} \leftarrow \varnothing$ , $\overline{\boldsymbol{X}} \leftarrow \varnothing$
2  $minMaxCost \leftarrow 1$ // based on Eq. 6, the maximum cost is 1
   // Initialize with CNNs that have the highest accuracy
3  $P \leftarrow \{\forall s$: $m_s | m_s \in M_s$ and has the highest accuracy$\}$
4  while $\exists m_s \in P$ is NOT with the lowest accuracy do
5      CFL $\leftarrow$ common frozen layers of CNNs in $P$
6      $\boldsymbol{Q} \leftarrow \varnothing$
7      for $m_s \in P$ do
           // split each model based on common frozen layers
8          $\boldsymbol{m_s} \leftarrow (m_s^1, m_s^2)$, separation of $m_s$ based on CFL
           // record separate models as model selection
9          $\boldsymbol{Q}.add(\boldsymbol{m_s})$
10     $\boldsymbol{X} \leftarrow$ TaskAssign($\boldsymbol{Q}$, Input) // Algorithm 2
11     $cost \leftarrow 0$, $exit \leftarrow ture$, $accCost \leftarrow 1$, $\overline{s} \leftarrow NULL$
12     for $s \in S$ do
           // calculate cost of each task and find the maximum
13         $cost \leftarrow \max(cost, C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s))$
           // update flag for early exit
14         if $L(\boldsymbol{m_s}, \boldsymbol{u_s^k}) > L_s$ or $T(\boldsymbol{m_s}, \boldsymbol{u_s^k}) < T_s$ then
15             $exit \leftarrow false$
           // find task $\overline{s}$ with the minimum accuracy cost
16         if $accCost > \alpha_s \cdot \frac{A_s - A(\boldsymbol{m_s})}{A_s}$ then
17             $\overline{s} \leftarrow s$
18             $accCost \leftarrow \alpha_s \cdot \frac{A_s - A(\boldsymbol{m_s})}{A_s}$
       // update minMaxCost, model selection and task assignment
19     if $cost < minMaxCost$ then
20         $minMaxCost \leftarrow cost$
21         $\overline{\boldsymbol{Q}} \leftarrow \boldsymbol{Q}$, $\overline{\boldsymbol{X}} \leftarrow \boldsymbol{X}$
22     if $exit = ture$ then
23         return $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$
       // replace model for task with the minimum accuracy cost
24     $m'_{\overline{s}} \leftarrow$ CNN in $M_{\overline{s}}$ with accuracy second to $m_{\overline{s}} \in P$
25     $P \leftarrow P \setminus \{m_{\overline{s}}\} \cup \{m'_{\overline{s}}\}$ // update model in $P$ for task $\overline{s}$
26 return $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$
```

Algorithm 2: TaskAssign(), a genetic algorithm

```
Input  : $\boldsymbol{Q} = \{\boldsymbol{m_s}\}$; ∀$s$: $A_s$, $L_s$, $T_s$, $\alpha_s$, $\beta_s$, $\gamma_s$;
         ∀$k, k'$: $U_k$, $R_k$, $b_{kk'}$; ∀$m_s^i$, $k$: $l_{m_s^i}^k, t_{m_s^i}^k$
Output: Task assignment $\boldsymbol{X}$
1  $P \leftarrow$ RandomlyInitPopulations()
2  $\boldsymbol{X} \leftarrow$ SelectBest$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P)$
3  $gen \leftarrow 0$;
4  while $gen < MAXGEN$ do
5      $P_{pa} \leftarrow$ SelectParents$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P)$
6      $P_{ch} \leftarrow$ GenerateChildren$(P_{pa})$
7      $P_{mu} \leftarrow P_{ch}$
8      for $p \in P_{mu}$ do
9          $p \leftarrow$ Mutate$(p, MR)$ // mutation rate: $MR$
           // recombination period: $RP$
10         if $gen\%RP = 0$ then
11             $p \leftarrow$ Recombination$(p)$
       // Filter constraints: $CSTR$
12     $P_{off} \leftarrow$ FilterOffspring$(CSTR, P_{ch} \cup P_{mu})$
13     $P \leftarrow$ SelectPopulations$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), P_{pa} \cup P_{off})$
14     $\boldsymbol{X} \leftarrow$ SelectBest$(C(\boldsymbol{m_s}, \boldsymbol{u_s^k}, s), \{\boldsymbol{X}\} \cup P)$
15 return $\boldsymbol{X}$
```

create a set P by choosing the model with the highest inference accuracy for each task (line 3). Starting with this initial set, a procedure is performed repeatedly until all the models in the set have the lowest accuracy (line 4). In the procedure, we first find the Common Frozen Layers (CFL) among all the models (line 5). Then, base on the common frozen layer, each model is split into two parts and stored in $\boldsymbol{Q}$ (line 6-9). Next, based on the separate models and the input information, we call a task assignment procedure (Algorithm 2) to get a task assignment $\boldsymbol{X}$ (line 10). Based on the assignment $\boldsymbol{X}$, we calculate the cost function of each task and record the maximum one (line 13). We also calculate the latency and throughput of each task and check if they meet the goals (line 14). If either of them does not meet the goal, the whole procedure cannot exit in advance (line 15). Besides, we also check which task has the minimum accuracy cost and record it for the later update (line 16-18). After we find the maximum cost of all tasks, we compare it with $minMaxCost$ and update $minMaxCost$ with new value it is smaller (line 19-20). Correspondingly, we also update $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ with a better solution (line 21). Then, we check if the procedure can exit early (line 22). If so, we get current $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ as the optimal solution (line 23). Otherwise, we update model set P by replacing the model of task with the minimum accuracy cost and continue the loop procedure (line 24-25). In the end, when all the models in P have the lowest accuracy, the loop procedure stops and the whole algorithm returns $\overline{\boldsymbol{Q}}$ and $\overline{\boldsymbol{X}}$ as the optimal solution (line 26).

Algorithm 2 describes the task assignment procedure used in Algorithm 1. It is a genetic algorithm which starts with a certain amount of random assignments (line 1) and runs an iterative process (line 4-14) containing the following operations: a) SelectParents, which selects a certain number of candidate assignments as parents based on the cost function (line 5); b) GenerateChildren, which generates children assignments by running uniform crossover (line 6); c) Mutate, which chooses some rows of an assignment and changes the values at some random positions (line 9); d) Recombination, which exchanges two rows of a task assignment to achieve a lower cost (line 11); e) FilterOffSpring, which deletes task assignments that violate constraints (line 12); f) SelectPopulations, which selects new populations from all the candidates based on the cost function; g) SelectBest, which selects an assignment with the minimum cost (line 13). After iterating the above operations for a certain

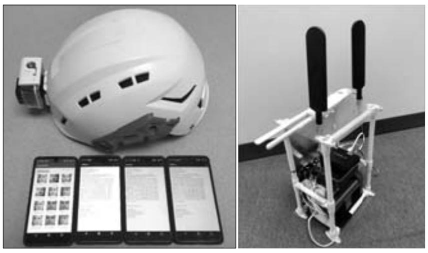


Fig. 8: A test platform for deploying *AMVP* which consists of a helmet camera, four Android phones and a wireless manpack

number of generation *gen*, it returns an assignment $\boldsymbol{X}$ which achieves the lowest cost with the best efforts (line 15).

A simple example of MSTA is shown in Fig. 7b, where three components (i.e., GenderNet, EmotionNet, AgeNet) of a video processing pipeline depend on CNNs. For each component, there are a set of CNN model candidates with two pre-trained CNN types (mobileNetV2, resNet50V2) and different frozen layers. Based on MSTA, GenderNet chooses a mobileNetV2-based CNN while both EmotionNet and AgeNet choose the resNet50V2-based CNNs. GenderNet runs the selected CNN at device M4. EmotionNet and AgeNet execute the common frozen layers of the selected CNNs at device M2 and execute the distinct layers of the selected CNNs at device M3.

V. Evaluation

A. Implementation and experimental setup

The implementation of *AMVP* are divided into two stages: offline stage and online stage. For the offline stage, we first train a group of CNNs for different vision analytic tasks (e.g., gender recognition, emotion recognition, age recognition, etc.) via transfer learning with Keras [43] API of TensorFlow [44]. These CNNs are trained from different pre-trained CNNs with different frozen layers, which result in a group of different .h5 models. The pre-trained CNNs we adopt are mobileNetV2 and resNet50V2 trained on ImageNet [45]. The dataset we use to train each vision task contains 750 images for each category: 500 images for training and 250 images for validation. After getting the .h5 models, we split each model at different layers and convert the separate models into .tflite format. This results in a group of .tflite model pairs. Next, we run each .tflite model pair on an Android phone to profile critical information such as inference accuracy, processing latency, memory footprint, feature traffic size, etc. We gather these information together as a database for the later model selection and assignment.

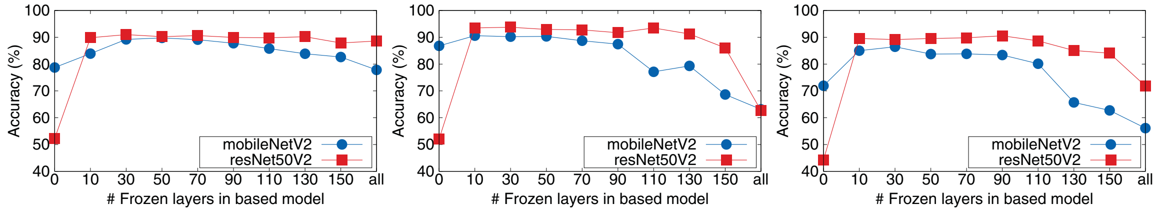
Next, for the online stage, we implement *AMVP* on top of a mobile distributed stream processing system [30] running on a test platform shown in Fig. 8, which consists of a helmet Yi® camera, four Essentail® phones and a wireless manpack. An *AMVP* client application with a group of .tflite model pairs is installed on each phone in advance. An *AMVP* server with the profile database is running on the manpack for model selection and task assignment. Before a user executes a multitask video processing application, he/she first sends a request containing the application information and the performance goals to the *AMVP* server. When the server receives the request, it calls the MSTA algorithm to select an appropriate CNN model for each vision task and assigns them to run on proper devices. When an *AMVP* client receives an assignment from the server, it loads the selected CNNs to memory to perform vision analysis.

In our experiments, we utilizes a multitask video processing application in Fig. 7b to evaluate *AMVP*. The first component C1 of this application pulls stream from the video source and chops the stream into frames and the second component C2 detects whether there is a face in each frame and outputs the frames containing faces into GenderNet, EmotionNet, AgeNet, respectively. The component C3 finally collects the inference results. We compare *AMVP* with two baseline strategies, i.e., Pure Sharing Strategy (PSS), which shares the common frozen layers among multiple CNNs on a single mobile device and Pure Offloading Strategy (POS), which offloads CNNs layers to other devices without sharing. We run extensive experiments with various performance goals and parameters under different computation and networking conditions to show how *AMVP* adapts to the different edge computing environment.

B. Experimental results

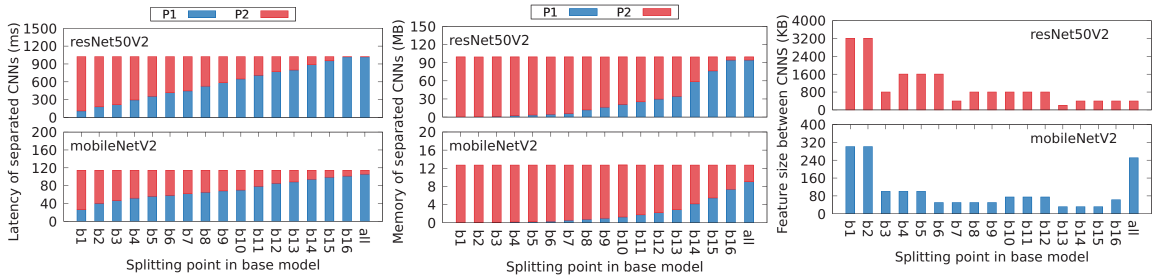
1) Transfer learning of CNNs: We train CNNs for different vision analysis tasks through transfer learning with different frozen layers in the pre-trained models. As shown in Fig. 9, in general, resNet50V2-based CNNs achieves higher inference accuracy than mobileNetV2-based CNNs because the former have deeper layers (190 vs. 155) and more parameters (23.56M vs. 2.25M). For each specific vision task, we observe that, as the number of frozen layers increases, the inference accuracy first increases and then decreases. This is because the datasets we use for training different vision tasks are much smaller than the ImageNet [45] dataset used for training resNet50V2 and mobileNetV2. If we retrain all the layers in the base model (Strategy 1 in Fig. 5), it is easy to encounter the overfitting problem. Specifically, we find that, the overfitting problem for resNet50V2-based CNNs is more serious than mobileNetV2-based CNNs because it has more parameters. By contrast, if we freeze all the layers in the based model (Strategy 2 in Fig. 5), it is difficult to extract specific features for each vision analysis task, which leads to a low accuracy as well. Therefore, freezing some part of the layers in the base model (Strategy 3 in Fig. 5) is a wise choice when performing transfer learning on large parameter CNNs with small dataset. Besides, we also observe that, the effect of freezing proper layers is more obvious for complicated applications. For example, as shown in Fig. 9a, for gender recognition which is simpler than emotion or age recognition, the accuracy difference of models with different frozen layers is much smaller.

2) CNN model profiling: As shown in Fig. 10, we choose 17 points as the candidate splitting points for mobileNetV2 and resNet50V2. All these splitting points are the last layers of CNN blocks. For each splitting point, we can divide a CNN model into a pair of separate CNN parts P1 and P2. For each pair of separate CNN parts, we profile the latency and memory size, as well as the feature size between them. As we see



(a) GenderNet (b) EmotionNet (c) AgeNet

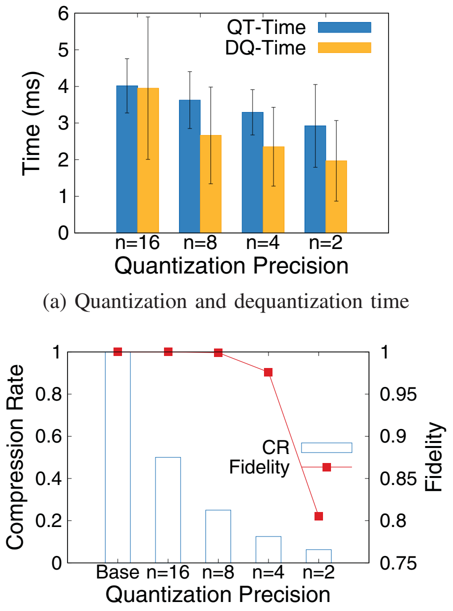
Fig. 9: The number of frozen layers in the pre-trained CNNs affects the inference accuracy of vision analysis tasks



(a) Latency (b) Memory size (c) Feature size

Fig. 10: Latency, memory size of and feature size between separate CNNs at different splitting points

in Fig. 10a, the latency of P1 almost increases linearly with the block number. Therefore, to balance the computation workload between P1 and P2, the optimal splitting point is in the middle. However, the memory size of a CNN is mainly concentrated in the back of the model (Fig. 10b). Meanwhile, the feature size between P1 and P2 decreases as the block number increases. Therefore, from the perspective of memory print balance and communication cost, splitting the model at a latter layer is better. Moreover, from the layer sharing perspective, if we split different CNNs at a latter layer, these CNNs will have a larger opportunity to share more layers by freezing more layers of their first parts. However, as we observe from Fig. 9, freezing too many CNN layers in the convolutional base decreases the inference accuracy of vision tasks. Overall, from the above profile data, we can conclude that, when we split a CNN model into two separate parts, there is a complicated trade-off among different metrics, which includes computation workload balance, memory footprint balance, communication costs, shareable common layers, and inference accuracy. This motivates us to design a framework that is able to adaptively choose the most appropriate splitting points for different CNNs based on the user requirements and environment condition.

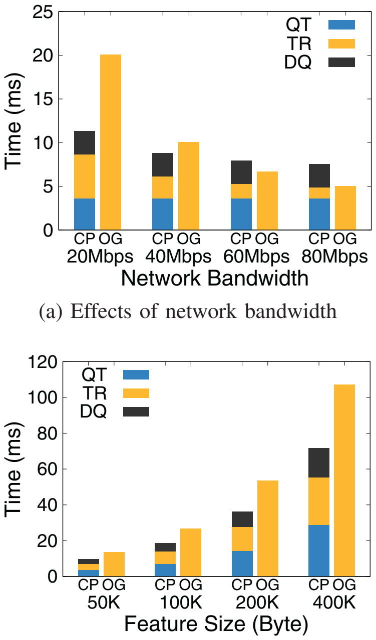


(a) Quantization and dequantization time

(b) Compression rate and fidelity

Fig. 11: An example (mobileNetV2-based AgeNet, splitting at layer 90, feature size = 14*14*64*4 ≈ 50 KB) that shows how quantization precision affects quantization/dequantization time (QT-Time/DQ-Time), compression rate (CR) and fidelity.

3) Feature compression and transmission: After we split a CNN into two separate parts P1 and P2 and deploy them on two devices, P1 needs to transmit its extracted feature to P2 to finish the inference. As mentioned in IV-B, in *AMVP*, we use a quantization-based method to compress the feature before transmitting it. In Fig. 11, we use a simple example to show how the quantization precision affects the total quantization and dequantization time, compression rate, and fidelity. We use a mobileNetV2-based AgeNet, splitting at layer 90, with inter-part feature size around 50KB. As we see from Fig. 11a, when the quantization precision n decreases, both quantization and dequantization time decreases. From Fig. 11b, we observe that,



(a) Effects of network bandwidth

(b) Effects of feature size

Fig. 12: An example that compares quantization, transmission, and dequantization time (QT, TR, and DQ) of our compression method (CP) and the original transmission (OG) with different network bandwidths and feature sizes. In (a), the feature size is fixed at 50 KB and the network bandwidth increases from 20Mbps to 80Mbps. In (b), the network bandwidth is fixed at 30Mbps and the feature size increases from 50KB to 400KB.

although the compression rate decreases as the quantization precision decreases, the fidelity metric starts to decrease at some point (n=8, fidelity=0.999) as well. To keep the inference accuracy, we use n=8 in *AMVP*.

In Fig. 12, we compare the quantization, transmission, and dequantization time (QT, TR, and DQ) of our compression method (CP) and the original transmission (OG) with different network bandwidths and feature sizes. In Fig. 12a, we transmit the feature with fixed size (50KB) at different network bandwidths. When the network bandwidth is low, even though (de)quantization takes some time, the total amount of time for delivering the feature is still lower than that of delivering it in original size. However, as the network bandwidth increases, transmitting feature in original size becomes a better choice because the (de)quantization time exceeds the saved transmission time. In Fig. 12b, we show that, when the network bandwidth is at a low level (30Mbps), the larger the feature size is, the more compression benefits (CPB) we can obtain by using the quantization-based compression. In *AMVP*, system can adaptively use the quantization-based compression method based on the network condition.

4) Adapting to different accuracy requirement: The user of *AMVP* may have different accuracy requirements. In Fig. 13, we show how *AMVP* adapts to different accuracy requirements by trading off other metrics. As we can see, the first user has an accuracy goal 80% for all three vision tasks, which is not very high. He also has a latency goal 1000ms and a throughput goal 1F/s. Since he does not have a strong preference on the accuracy goal, he sets α, β, γ all equal to 0.33. The second user, however, has a much higher accuracy goal 90% and he has a latency goal 1000ms and a throughput goal 1F/s as well. Since he has strong preference on fulfilling the accuracy goal, he sets $\alpha = 0.8$, $\beta = 0.1$, and $\gamma = 0.1$, respectively. Based on the different accuracy goals and preference, *AMVP* offers a solution with lower accuracy and lower latency for the first user. For the second user, however, *AMVP* offers a solution with higher accuracy by trading off the latency. Both solutions meet the throughput goals.

5) Adapt to different throughput requirement: Video stream can be fed into *AMVP* at different frame rate, which requires *AMVP* to provide different throughput. As shown in Fig. 14, when the input rate is 1F/s, *AMVP* provides a solution with high accuracy by using the resNet50V2-based model for each task, which also has higher latency. However, when the input rate becomes 3F/s, *AMVP* provides another solution by using the mobileNetV2-based models. This solution provides 2.8x higher throughput and 5x lower latency than the original.

6) Adapting to different latency requirement: Sometimes, an *AMVP* user wants to instantly get an inference result, which requires *AMVP* to provide a short latency. As shown in Fig. 15, when the latency required by the user is high (1500ms), *AMVP* provides a solution with a high latency and a high accuracy. However, once the latency requirement becomes low (200ms), *AMVP* will choose another solution with lower latency, as well as lower accuracy. The throughput in both solutions are equal to the input rate.

7) Adapt to different computing resource: The actual available computing resource of a mobile device at the edge is uncertain because other applications also consume it. In Fig. 16, we show how *AMVP* adapts to computing resource change of a mobile device by running a synthetic workload application to consume some computing resources in the background. We compare *AMVP* with two baseline strategies, i.e., Pure Sharing Strategy (PSS), which shares common layers among CNNs on a single device, and Pure Offloading Strategy (POS), which offloads CNNs to other devices, to show the superiority of *AMVP*. As we see in the left figure, comparing with scenarios with no workload application ("-no"), in the scenarios with workload application ("-wl"), all the strategies choose to run CNNs with lower accuracy to reduce the total computation cost. From the middle figure, we observe that: (a) when there is no workload, *AMVP* achieves up to 48% shorter latency than PSS by offloading some CNN layers to other devices and around 6% shorter latency than POS by sharing some common layers among CNNs; (b) when there is workload, although all the three strategies choose a lightweight mobileNetV2-based model for each task, *AMVP* achieves up to 61% shorter latency than PSS and POS; (c) for POS in the workload running case, GenderNet running on the mobile device with the workload has much higher latency than EmotionNet and AgeNet running

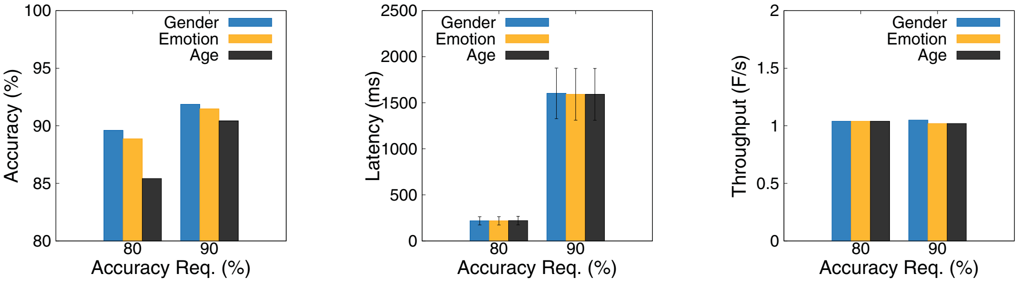


Fig. 13: *AMVP* adapts to different accuracy requirement.

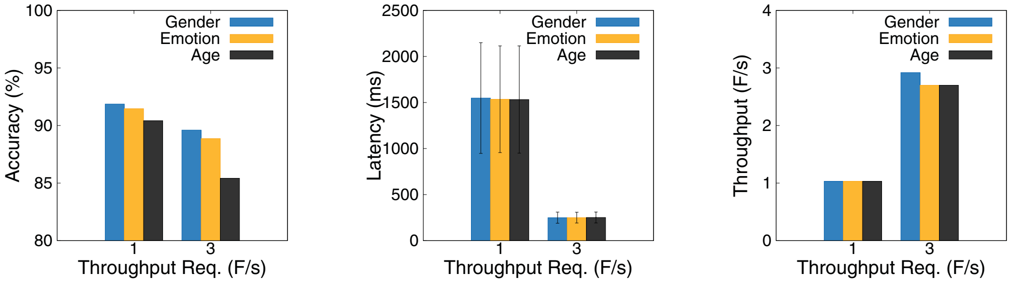


Fig. 14: *AMVP* adapts to different throughput requirement.

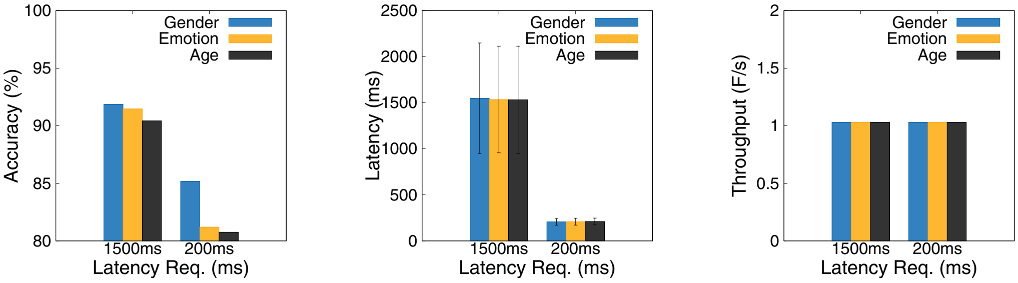


Fig. 15: *AMVP* adapts to different latency requirement.

on other devices that are idle. From the right figure, we see that, when there is no workload, all three strategies achieve similar throughput. However, when there is workload running, *AMVP* achieves around 10% higher throughput than PSS and around 7% higher throughput than POS.

8) Adapt to different network bandwidth: The edge network is dynamic. In Fig. 17, we show how *AMVP* outperforms POS by running application under different network bandwidths. As we observe in the left figure, when the network bandwidth drops from 50Mbps to 1Mbps, *AMVP* sacrifices the accuracy of EmotionNet by selecting a model with more frozen layers so that different CNNs can share more layers and all execute locally. However, POS still uses the original models because it does not care about layer sharing. From the middle figure, we observe that, when the network bandwidth becomes 1Mbps, *AMVP* executes all the CNNs locally while POS still offloads two CNNs to other devices, which causes larger communication latency. From the right figure, we find that, *AMVP* achieves around 1T/s throughput for all tasks in both 50Mbps and 1Mbps scenarios. However, since POS offloads two tasks to other devices when the network bandwidth is 1Mbps, the throughput of those offloaded tasks is restrict by the network bandwidth and thus decreases.

VI. Related Work

There are plenty of existing works regarding how to execute computational intensive CNNs on resource constrained mobile devices to support vision analysis. We classify these works into three categories and describe them from high level as follows.

CNN offloading. The key idea of CNN offloading is moving some CNN layers or the whole CNN model from the resource constrained mobile devices to some resource sufficient servers, no matter the server is in the cloud or at the edge. MCDNN [8] is an earlier representative work which deploys different CNN variants on both cloud and mobile devices. The variants at the cloud have higher accuracy but higher computation cost and extra communication overhead. The variants at mobile devices have lower computation cost and no communication overhead but lower accuracy. MCDNN selects a proper CNN variant at runtime to adapt to the dynamic operating conditions. Another representative work of CNN offloading is Neurosurgeon [46], which performs CNN computation partition between mobile

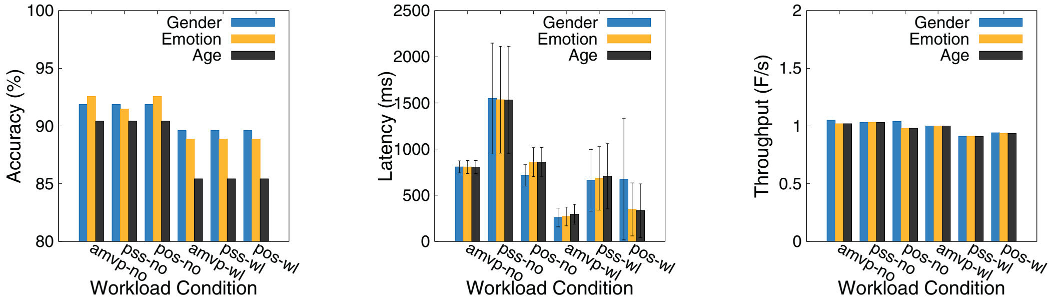


Fig. 16: Comparing *AMVP* with pure sharing strategy (pss) and pure offloading strategy (pos) with different computing resources. "-no" indicates that there is no other application with heavy workload running in the background, i.e., there are more computing resources for the video processing application. In contrast, "-wl" indicates that there is other application with heavy workload running in the background, i.e., there are fewer computing resources for the video processing application.

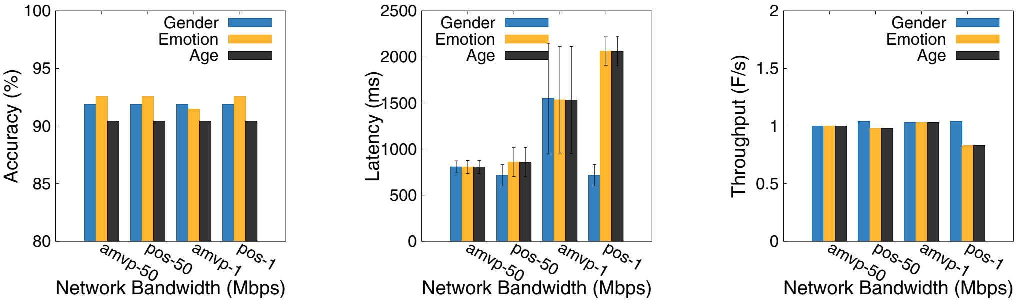


Fig. 17: Comparing *AMVP* with pure offloading strategy (pos) under different network bandwidths, where ”-1” indicates that the bandwidth is 1Mbps and ”-50” indicates that the bandwidth is 50Mbps.

devices and cloud at the granularity of neural network layers. It demonstrates that, compared to those cloud-only solutions, this collaborative solution achieves lower latency, lower energy consumption and higher datacenter throughput. A recent work named as Couper [12] brings edge server into CNN offloading by quickly slicing CNNs into components executing on both the cloud and edge. It proves via extensive experiments that a powerful edge server is essential for CNN offloading when the required latency is low while the network condition to the cloud is bad. Although the above solutions achieve significant performance improvement, they either rely on the cloud or a powerful edge server. Under some extreme conditions without Internet or powerful edge server, such solutions will fail.

Recent works also start to study distributing CNN execution on several IoT or mobile devices. Modnn [14] and Mednn [15] utilize specific partition schemes to partition CNN models onto several mobile devices to alleviate device-level computing cost and memory footprint. DeepThings [17] proposes distributed inference on IoT devices by employing a fused tile partitioning of convolution layers to expose parallelism, a distributed work stealing approach to balance dynamic workload and a novel scheduling procedure to reduce the overall execution latency. Musical Chair [18] supports efficient recognition at local by harvesting aggregated computational power from IoT devices in the same network. It explores both data parallelism and model parallelism of CNN to deal with the inherit dynamic. Different from above works which focus on the execution of single CNN, *AMVP* studies how to run multiple CNNs on the mobile devices, which is obviously more challenging.

CNN compression. CNN compression uses different compression techniques to construct efficient CNN models for mobile devices. XNOR-Net [47] approximates both input and weights into binary values to reduce the computation workload for real-time inference at the cost of some accuracy loss. ThiNet [48] applies filter level pruning to compresses CNNs by greedily pruning the filter that has the minimum effect on the activation values of the next layer. Factorized Networks [22] factorizes a high-cost 3D convolution operation as a low-cost single intra-channel convolution and a linear channel projection to reduce the computation while maintain the accuracy. All these works, however, have an identical limitation: they use a fixed compression technique to compress a CNN, which results in a one-for-all model that cannot adapt to different performance goals and resource constraints. To deal with this issue, a recent work [25] enables on-demand model compression by applying proper compression techniques to different CNN layers, which achieves an optimal balance between performance goals and resource constraints. In current *AMVP*, there is no direct model compression technique to support CNN model compression. However, in the future, we plan to integrate some.

Except for the above works which compress CNN layers to reduce computation cost and memory footprint, there are also some other works which compress the intermediate features to reduce the feature transmission size. For example, the authors in [40] claim that intermediate deep feature compression will become the next battlefield of collaborative intelligent sensing. In [37] and [41], the authors study the impact of lossy and near-losses feature compression on object detection accuracy. In [35], the authors present a lossy compression framework for intermediate deep feature compression based on quantization and video codec, which is adopted by the Audio Video Coding Standard Workgroup as the visual feature coding standard. In *AMVP*, in order to reduce the data traffic size of feature transmission between separate CNN components, we use similar methods to compress the feature. And in order to adapt to the uncertain edge network, we provide configures with different compression rate to provide trade-off between feature size and accuracy.

CNN sharing The key idea of CNN sharing is to share some common layers or parameters among multiple related CNNs to reduce the total computation workload or memory footprint. For example, in NestDNN [26], the authors propose a nesting method to allow different variants of a deep learning model to share common parameters, so that it can dynamically select the optimal resource-accuracy trade-off at runtime to fit each model's resource demand to the system available resources. In Mainstream [9], the authors propose to share some common layers among multiple CNNs at an edge server to reduce the total computation workload. At deployment time, based on the available resources and mix of applications on the edge server, it automatically determines the right trade-off between per-frame accuracy and more frames per second by choosing different number of shared layers. *AMVP* adopts similar layer sharing strategy as Mainstream. The difference is, instead of choosing proper shared layers based on the available resource at edge server, *AMVP* needs to consider the available resources of a mobile device cluster, as well as the dynamic bandwidth of the wireless network connecting them.

VII. Conclusion and Future Work

In this paper, we describe the design, implementation and evaluation of *AMVP*, an adaptive execution framework which supports CNN-based multitask video processing on a mobile distributed stream processing platform. *AMVP* combines the strategy of model sharing and model offloading, which enables multiple closely-related CNNs to share common frozen layers to reduce the total computation cost and allows one mobile device to offload part of CNN models to nearby mobile devices to reduce the local computation workload. To adapt to different performance goals and uncertain system status, the number of CNN layers for sharing and offloading is determined online. We conduct extensive experiments to show the superiority of *AMVP*. The experimental results show that, compared to those baseline solutions, *AMVP* achieves up to 61% lower latency and 10% higher throughput with comparative accuracy.

Current *AMVP* still have some limitations. First, it adopts a static accuracy metric, which does not meet the requirement of video analytics very well. For the future work, we consider applying similar accuracy metrics as in Chameleon [49], which includes precision, recall and F1 score. Second, current *AMVP* only executes on a mobile stream processing platform. For the future, we plan to generalize it to deploy on a heterogeneous stream processing platform, which leverages all the available resources including mobile devices, edge servers and cloud to perform CNN-based multitask video processing. Third, current *AMVP* only supports single video processing, evaluation with simultaneous processing of multiple videos will be conducted in the future. Finally, *AMVP* assumes that CNNs of different tasks are trained from the same base networks. Although this is a premise for sharing common layers among different CNNs, it is also a limitation of our solution.

Acknowledgement

This material is based upon a project supported by National Institute of Standards and Technology (NIST) under Grant NO. (#70NANB17H190). We also appreciate the suggestions from the anonymous reviewers and our shepherd Dr. Mi Zhang.

References

[1] K. Muhammad, J. Ahmad, Z. Lv, P. Bellavista, P. Yang, and S. W. Baik, "Efficient deep cnn-based fire detection and localization in video surveillance applications," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 7, pp. 1419–1434, 2018.

[2] M. Ravanbakhsh, M. Nabi, H. Mousavi, E. Sangineto, and N. Sebe, "Plug-and-play cnn for crowd motion analysis: An application in abnormal event detection," in *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018, pp. 1689–1698.

[3] M. Babaee, D. T. Dinh, and G. Rigoll, "A deep convolutional neural network for video sequence background subtraction," *Pattern Recognition*, vol. 76, pp. 635–649, 2018.

[4] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors," in *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*. ACM, 2019, p. 54.

[5] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Processing Letters*, vol. 23, no. 10, pp. 1499–1503, 2016.

[6] R. Ranjan, V. M. Patel, and R. Chellappa, "Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 1, pp. 121–135, 2017.

[7] D. C. Luvizon, D. Picard, and H. Tabia, "2d/3d pose estimation and action recognition using multitask deep learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5137–5146.

[8] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016, pp. 123–136.

[9] A. Jiang, D. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. Kozuch, P. Pillai, and G. Andersen, Davidand Ganger, "Mainstream: Dynamic stem-sharing for multi-tenant video processing," in *Proceedings of 2018 USENIX Annual Technical Conference*, 2018, pp. 29–42.

[10] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "Videochef: efficient approximation for streaming video processing pipelines," in *Proceedings of 2018 USENIX Annual Technical Conference*, 2018, pp. 43–56.

[11] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, G. Maghanath, and S. Bagchi, "Approxnet: Content and contention aware video analytics system for the edge," *arXiv preprint arXiv:1909.02068*, 2019.

[12] K.-J. Hsu, K. Bhardwaj, and A. Gavrilovska, "Couper: Dnn model slicing for visual analytics containers at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 179–194.

[13] S. M. George, W. Zhou, H. Chenji, M. Won, Y. O. Lee, A. Pazarloglou, R. Stoleru, and P. Barooah, "Distressnet: a wireless ad hoc and sensor network architecture for situation management in disaster response," *IEEE Communications Magazine*, vol. 48, no. 3, pp. 128–136, 2010.

[14] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Proceedings of 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1396–1401.

[15] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE, 2017, pp. 751–756.

[16] Z. Xu, Z. Qin, F. Yu, C. Liu, and X. Chen, "Direct: Resource-aware dynamic model reconfiguration for convolutional neural network in mobile systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2018, p. 37.

[17] 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.

[18] R. Hadidi, J. Cao, M. Woodward, M. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative iot devices," *arXiv preprint arXiv:1802.02138*, 2018.

[19] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[20] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[21] 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.

[22] M. Wang, B. Liu, and H. Foroosh, "Factorized convolutional neural networks," in *ICCV Workshops*, 2017, pp. 545–553.

[23] 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.

[24] S. Changpinyo, M. Sandler, and A. Zhmoginov, "The power of sparsity in convolutional neural networks," *arXiv preprint:1702.06257*, 2017.

[25] 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*, 2018, pp. 389–400.

[26] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2018, pp. 115–127.

[27] 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.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[29] Q. Ning, C.-A. Chen, R. Stoleru, and C. Chen, "Mobile storm: Distributed real-time stream processing for mobile clouds," in *Proceedings of 2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*. IEEE, 2015, pp. 139–145.

[30] M. Chao, C. Yang, Y. Zeng, and R. Stoleru, "F-mstorm: Feedback-based online distributed mobile stream processing," in *Proceedings of 2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 273–285.

[31] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: resilient edge processing for the internet of things," *VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[32] S. Fan, T. Salonidis, and B. Lee, "Swing: Swarm computing for mobile sensing," in *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1107–1117.

[33] M. Chao and R. Stoleru, "R-mstorm: A resilient mobile stream processing system for dynamic edge networks," in *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2020, pp. 64–72.

[34] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[35] Z. Chen, K. Fan, S. Wang, L.-Y. Duan, W. Lin, and A. Kot, "Lossy intermediate deep learning feature compression and evaluation," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 2414–2422.

[36] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein, "Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019, pp. 27–32.

[37] H. Choi and I. V. Bajić, "Deep feature compression for collaborative object detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2018, pp. 3743–3747.

[38] S. Luo, Y. Yang, Y. Yin, C. Shen, Y. Zhao, and M. Song, "Deepsic: Deep semantic image compression," in *International Conference on Neural Information Processing*. Springer, 2018, pp. 96–106.

[39] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: an efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, 2019.

[40] Z. Chen, W. Lin, S. Wang, L. Duan, and A. C. Kot, "Intermediate deep feature compression: the next battlefield of intelligent sensing," *arXiv preprint arXiv:1809.06196*, 2018.

[41] H. Choi and I. V. Bajić, "Near-lossless deep feature compression for collaborative intelligence," in *Proceedings of 20th IEEE International Workshop on Multimedia Signal Processing (MMSP)*, 2018, pp. 1–6.

[42] Z. Chen, K. Fan, S. Wang, L. Duan, W. Lin, and A. C. Kot, "Toward intelligent sensing: Intermediate deep feature compression," *IEEE Transactions on Image Processing*, vol. 29, pp. 2230–2243, 2019.

[43] "Keras: The python deep learning library," https://keras.io/, accessed: 2020-04-10.

[44] "Tensorflow: An end-to-end open source machine learning platform," https://www.tensorflow.org/, accessed: 2020-04-10.

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

[46] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[47] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Proceedings of European Conference on Computer Vision*, 2016, pp. 525–542.

[48] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[49] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 253–266.