Deep Learning Video Analytics on Edge Computing Devices

Tianxiang Tan and Guohong Cao
Department of Computer Science and Engineering
The Pennsylvania State University
Email: {txt51, gxc27}@psu.edu

Abstract—**The rapid progress of deep learning-based techniques such as Convolutional Neural Network (CNN) has enabled many emerging applications related to video analytics and running them on mobile devices can help improve our daily lives in many ways. However, there are many challenges for video analytics on mobile devices using multiple CNN models. CNN models are resource hungry, and each model requires a large amount of computational power and occupies a large portion of memory space. Although video processing can be offloaded to reduce the computation time, transmitting large amount of video data is time consuming. Thus, offloading is not always the best option. Moreover, different CNN models have different memory usage and processing time, making the scheduling problem more complex. As a result, besides deciding which task to be offloaded, we must decide which CNN model should reside in the memory and for how long, and which CNN model should be switched out due to memory constraint. In this paper, we propose resource aware scheduling algorithms to address these challenges. We identify the task scheduling problem for running multiple CNN models on mobile devices under resource constraints and formulate it as an integer programming problem. We propose resource-aware scheduling algorithms which combine offloading and local processing methods to minimize the completion time of video processing. We implement the proposed scheduling algorithms on Android-based smartphones and demonstrate its effectiveness through extensive experiments.**

I. Introduction

Over the past few years, there has been significant progress in deep learning-based techniques such as Convolutional Neural Network (CNN), and researchers have been applying these techniques to solve computer vision and natural language processing problems [1], [2]. These techniques can provide much better results than traditional methods and some of them even outperform human beings in specific datasets [3].

With the help of CNN models, more intelligent analytics can be performed on videos captured by mobile devices. Video analytic usually includes two phases: detecting objects in the video frames and recognizing them. The extracted information from the captured video can help improve our daily lives in many ways. For example, a traveler with a Google Glass can continuously capture images of different people, objects, or even street signs written in a foreign language. Multiple CNN models are applied to recognize them to improve the accuracy since each CNN model is only good at recognizing some class of objects. For example, buildings are recognized with PlacesCNN [4] so that the traveler can quickly recognize the well-known landmarks and never miss important tour points. Human faces are recognized with VGGNet [5] in case an old friend or a celebrity shows up, and GoogleNet is used to detect the words in street signs, restaurant menus, and replace them with translated texts if necessary, so that the traveler can quickly adapt to a new environment. Such video analytics are also useful for content-based indexing, which can help people find the necessary information quickly; otherwise, all videos have to be searched to find a particular object. It can also help people with memory disorder to restore their memories by retrieving the relevant video collected earlier [6].

There are many challenges for video analytics on mobile devices using multiple CNN models. CNN models are resource hungry, and each model requires a large amount of computational power and occupies a large portion of memory space. The problem becomes worse for mobile devices which have limited resources and the time for running a CNN model on mobile device is usually long. For example, it takes about 900ms to use a pre-trained Resnet model to process one image on Samsung S8 with the Android Caffe library, and it occupies more than 700 MB memory space to hold its parameters and intermediate result during execution. Moreover, when multiple CNN models are needed, the memory space will become a problem. For example, the available memory space is usually less than 1.3 GB on advanced smartphones like Samsung S8, since the operating system will occupy some memory space. As a result, to run multiple CNN models locally, memory switch is necessary since a mobile device cannot hold all CNN models in memory at the same time. Switching the models in/out of the memory is time consuming, and hence the scheduler should minimize such memory switches.

Videos can be offloaded to the edge server for processing. Although the server can reduce the computation time, it takes much time to transmit the videos which are usually large and mobile devices have limited wireless bandwidth. We can first detect the objects and only offload the extracted images from the video frames, but the mount of transmitted data may still be high. Thus, offloading is not always the best option. Moreover, different CNN models have different memory usage and processing time, making the scheduling problem more complex. As a result, besides deciding which task to be offloaded, we have to decide which CNN model should reside in the memory and for how long, and which CNN model should be switched out at which time. Although researchers have

978-1-6654-4108-7/21/$31.00 ©2021 IEEE

proposed offloading techniques [7]–[9] which can determine jobs to be offloaded to the edge server or processed locally to minimize the completion time, none of them considers the memory constraint and its effect on scheduling. Recently, there is some work [10], [11] on using multiple versions of the same CNN model to achieve tradeoffs between processing time and accuracy. However, none of them considers running multiple CNN models or design scheduling algorithms to minimize the overall video processing time.

In this paper, we propose resource aware scheduling algorithms to address these challenges. Our goal is to minimize the completion time of video processing, which is accomplished by first deciding whether an incoming image processing task should be offloaded or executed locally. If a task is decided to be executed locally, its corresponding CNN model must be loaded into the memory, possibly by switching out some other models due to memory constraint. We formulate this scheduling problem as an integer programming problem and propose two heuristic based algorithms: a naive algorithm which decides whether to offload or run locally based on their completion time difference, and an advanced algorithm which addresses some weaknesses of the naive algorithm to further reduce the completion time.

Our contributions can be summarized as follows.

- We identify the task scheduling problem for running multiple CNN models on mobile devices under resource constraints, and formulate it as an integer programming problem.
- We propose resource-aware scheduling algorithms which combine offloading and local processing methods to minimize the completion time of video processing.
- We implement the proposed scheduling algorithms on Android-based smartphones and demonstrate its effectiveness through extensive evaluations.

The organization of the paper is as follow: We discuss related work in Section II. In Section III, we give the motivation and the basic idea of our solution. We formulate the scheduling problem in Section IV, and present the proposed resource-aware scheduling algorithms in Section V. Section VI presents the evaluation results, and Section VII concludes the paper.

II. Related Work

There are some existing researches on supporting CNN models on mobile devices. Many of them focus on optimizing the CNN model with techniques such as model compression to reduce its resource requirement and execution time. Teerapittayanon *et al.* [12] inserts an early exit point to a CNN model, so that less accurate results can be obtained by running part of the model to reduce the completion time. Tan *et al.* [13] leverages Neural Processing Unit (NPU) and model partition techniques to improve the performance of running CNN models on mobile devices. Chameleon [14] achieve a better tradeoff between accuracy and processing time by dynamically changing the configuration of DNN used in video analytics. Liu *et al.* [15] not only considers how to reduce the network size, but also proposes methods to guarantee that the compressed model can satisfy the performance requirement.

Another widely used technique to improve the performance of video analytics is computation offloading, and there is considerable amount of existing research on computation offloading. For example, MAUI [7] profiles the energy consumption, offloading latency and task dependency for each function, and it uses offloading to optimize the energy usage. Computation offloading techniques [16], [17] have also been proposed to save energy by considering the long tail problem in 4G/LTE network, and by considering the energy performance tradeoff in multicore-based mobile devices. Some researchers studied how to satisfy the delay constraints by running different DNNs locally under various network conditions [10], [18]. DeepDecision [10] and MCDNN [11] aim to achieve the tradeoff between video processing speed and accuracy by using different versions of the same CNN model to process images. FastVA [18] leveraged Neural Processing Unit (NPU) and offloading techniques for video analytics on mobile devices. However, none of them considers the memory constraint of mobile devices and the characteristics of different CNN models.

Multiple CNN models have been considered in DeepEye [19], which uses separate threads to run the fully connected and convolutional layers, and it interleaves the execution of different layers so that multiple models can be executed on wearable devices with limited memory. However, DeepEye only runs CNN models locally, while our work considers offloading some images to the server. As a result, besides deciding which task to be offloaded, we must decide which CNN model should reside in the memory and for how long, and which CNN model should be switched out at which time.

III. Preliminary

Our goal is to find an efficient job schedule for video analytic using multiple CNN models on mobile devices. In order to achieve this goal, we first need to understand the CNN models. Although different CNN models share some common characteristics such as similar types of layers, there are many differences among them when considering their memory usage, loading time and processing time. Such difference plays an important role in making scheduling decisions, since a mobile device may not have enough memory to hold all the CNN models and some models can be executed more efficiently locally than others. In this section, we first introduce the general procedure of video analytic. Then we discuss the common layers in CNN models, which are directly related to the characteristics of CNN models. Finally, we present the motivation and general idea of our solution.

A. Video Analytic

There are two stages to process a video frame, which is shown in Fig. 1. The first one is object detection, where an object detector is used to locate the objects and classify the objects into different categories. Some methods [20], [21] have been proposed to do the detection in real time. For each frame,

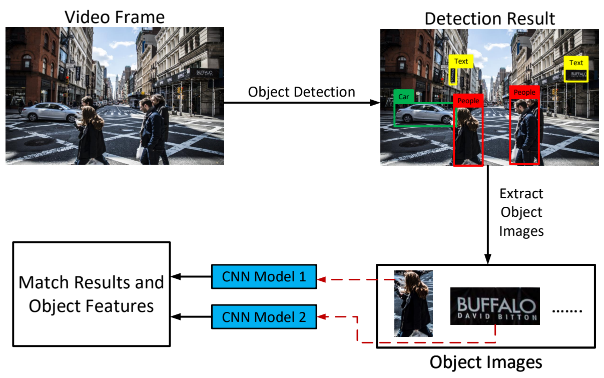


Fig. 1: The procedure of detecting and recognizing objects for a video frame.

the object detector will find out the location for each object and crop its image from the frame. Based on the detection result, the number of objects in each frame is obtained. The second stage is object recognition, in which the cropped object images are sent to the corresponding CNN models for extracting features and matching. Multiple CNN models are needed in this stage to improve the accuracy since each CNN model is only good at recognizing some class of objects.

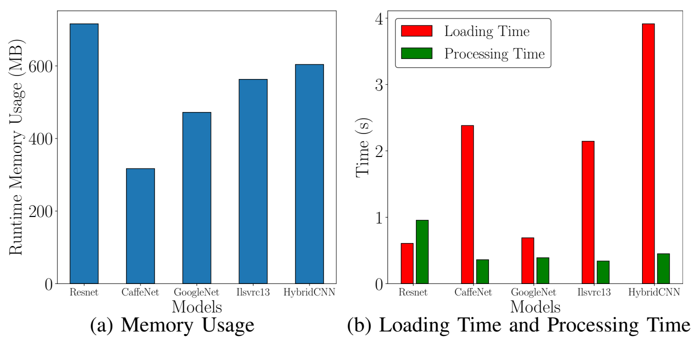
We focus on minimizing the completion time for feature extraction using different CNN models, since most of the video processing time is in this stage. As in the examples introduced in the introduction, reducing the video processing time will help reduce the user waiting time and assist users adapt to the new environment quickly or find the necessary information quickly. Many similar examples can be made for law enforcement where completion time is important.

A mobile device can choose to process an object image locally or offload it to the edge server. A good scheduler decides which images are offloaded to the edge server and lets the device to run the rest of them locally, so that the completion time can be minimized. We try to find out such a schedule given the CNN models, memory limitations, object image sizes and the wireless bandwidth.

B. Convolutional Neural Network

During video processing, different CNN models are used for extracting information from the object images. Although many CNN models have been proposed, most of them contain some common types of layers. Here, we only discuss the convolutional layer and the fully connected layer, which generate most of the processing time and loading time.

Convolutional Layer: The convolutional layers are located at the beginning of the CNN models which include dozens of filters. These filters are used to extract features from the input image. The convolution operations are performed when a filter is applied on the input data. Although storing the parameters for these filters does not need too much memory, performing the convolution operation requires lots of memory. Meanwhile, the intermediate results produced by this layer occupy a large amount of memory space. Since CNN models usually construct their convolution layers with different settings, the processing time and the memory usage spent on their convolutional layers are different.



(a) Memory Usage (b) Loading Time and Processing Time

Fig. 2: Memory space, loading time and processing time requirement of different CNN models for Samsung S8

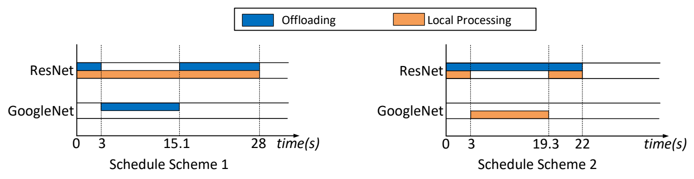


Fig. 3: An example of different schedules

Fully Connected Layer: The fully connected layers are usually located at the end of the CNN model. Normally, there are millions of parameters needed in a fully connected layer. Although the computation of fully connected layers can be completed quickly, the memory space required to hold the parameters is very large and most of the loading time for a CNN model will be spent on reading these parameters.

C. Motivation and Basic Idea

There are many CNN models. Some of them are frequently used and have been fine-tuned for different tasks. For example, the googleNet proposed in [3] is fine-tuned to recognize places in images [22]. Here we choose several well-known models from Caffe Model Zoo, which have been adopted by many researchers. To understand the characteristics of different CNN models, we measure the memory usage, loading time and processing time of the selected CNN models, since they directly affect the job scheduling decision. The deep learning framework we use is Caffe [23], and we export it to Android for evaluation on a Samsung S8. The measurement result is shown in Fig. 2.

As shown in Fig. 2(a), some CNN models require much more memory space than others. Since mobile devices have limited memory, it is impossible to hold all CNN models in memory. For instance, a device with 1 GB available memory space can either run an instance of Resnet or execute CaffeNet and GoogleNet simultaneously. These choices will lead to different completion time of video processing. Fig. 2(b) shows the loading time and the processing time of different CNN models. As can be seen, some models can process their data more efficiently than others, while some of them have a faster loading time. All these differences will affect the completion time of video processing, and we need to carefully decide which jobs should be processed locally and which of them should be offloaded. To see how these CNN models affect the job scheduling decision, we use an example to show the completion time difference of two schedule schemes in Fig. 3.

In Fig. 3, suppose 40 images need to be processed by ResNet at time $t = 0s$. At time $t = 3s$, 40 more images need to be processed by ResNet and 40 images need to be processed by GoogleNet. For simplicity, we assume all input data are 227x227 pixel RGB images, which are the inputs for all CNN models as well. We assume that the uplink bandwidth is 4Mbps and the available memory space on the mobile device is 1 GB.

We show two different schedules in Fig. 3. The yellow line means that the CNN model is currently used by the mobile device for processing images, while the blue line means that the CNN model is not in memory and its input images are sent to the edge server for processing. At the beginning, since there are only input images for Resnet, the mobile device loads the Resnet model into memory to process the data. Meanwhile, some of the input images are transmitted to the edge server in parallel to reduce the completion time. At $t = 3s$, there are jobs coming for both Resnet and GoolgeNet. At this point, scheme-1 chooses to keep Resnet in memory based on the intuition that model switching is time consuming, and it offloads the new arrived images for GoogleNet to the edge server. On the other hand, scheme-2 decides to switch out Resnet in order to run GoogleNet model on the mobile device, since Resnet is not as efficient as GoogleNet when being run locally.

From Fig. 3, we can see that scheme-2 can save 25% of time compared with scheme-1. Although this example is simple, in many real cases, there are many CNN models, and it is hard to decide which model should be run locally and which images should be offloaded. The decision is affected by many factors, and we will formulate and study it in the following sections.

IV. Problem Formulation

Table I shows the notations used in the problem formulation. For an image I_i, it can be processed locally or offloaded.

Notation	Meaning
I_i	The i^{th} Image
c_i	The CNN model used to process I_i locally
s_i	The data size of Image I_i
L_j	The loading time of the j^{th} CNN model
P_j	Local processing time of one image using the j^{th} model
m_j	The memory requirement to run the j^{th} CNN model
t_i	The time when I_i is processed or offloaded
M	The total available memory space of the device
B	The uplink bandwidth

TABLE I: Notations

Local Processing: If I_i is processed locally, its completion time can be calculated as $L_{c_i}X_{c_i}^{t_i} + P_{c_i} + t_i$, where $X_{c_i}^{t_i}$ is 1 if the c_i^{th} CNN model is in memory at time t_i and it is 0 otherwise. There are several constraints. First, the total memory usage of all CNN models loaded into the memory should not be greater than M at any given time; i.e., $\forall t, \sum_j m_j X_j^t \leq M$.

Second, for the same CNN model, it cannot process I_i before it finishes the processing of the images that arrive earlier, i.e., $t_i \geq t_{i'} + L_{c_{i'}}X_{c_{i'}}^{t_{i'}} + P_{c_{i'}}, \forall I_{i'}$, where $i' < i$ and $c_{i'} = c_i$.

Third, the CNN model must be kept in memory when we use it to process image I_i locally, i.e., during time $t' \in (t_i, t_i + L_{c_i}X_{c_i}^{t_i} + P_{c_i})$, $X_{c_i}^{t'} = 1$.

Offloading: to offload I_i, the completion time is $t_i + \frac{s_i}{B}$. The only constraint for offloading is that the network interface can only start offloading I_i after transmitting all the images that have arrived before I_i. That is, $t_i \geq t_{i'} + \frac{s_{i'}}{B}$, for all offloaded image $I_{i'}$, where $i' < i$.

We do not consider the time for the server to process and return results. The server inference time and the result returning time are negligible due to the following. After the mobile device offloads an image to the server, it does not need the result from the server to offload or process the next image. As a result, the server processing is not the bottleneck for calculating the completion time, and it can be processed in parallel with the (next image) uploading or local processing, and thus is ignored. Similarly, transmitting the processing results (several bytes for each image) is not the bottleneck and can be done in parallel with the uploading or local processing, and thus it is ignored.

Let e_{max}^{loc} denote the completion time for the last image to be processed locally, and then $e_{max}^{loc} = \max_i L_{c_i}X_{c_i}^{t_i} + P_{c_i} + t_i$. Let e_{max}^{off} denote the completion time of the last offloaded image, and then $e_{max}^{off} = \max_i \frac{s_i}{B} + t_i$. Let e_{max} denote the completion time of the last image, and it can be calculated as $\max(e_{max}^{loc}, e_{max}^{off})$. In fact, e_{max} is the time when we finish processing the video, and our goal is to minimize e_{max}. Therefore, the problem can be formulated as an integer programming in the following way:

$$
\begin{aligned}
\min \quad & e_{max} \\
\text{s.t} \quad & M \geq \sum_j m_j X_j^t, \forall t \in (0, e_{max}) \\
& X_{c_i}^{t'} \geq y_i, \forall t' \in (t_i, t_i + L_{c_i}X_{c_i}^{t_i} + P_{c_i}) \\
& t_i \geq (y_i y_{i'})(t_{i'} + L_{c_{i'}}X_{c_{i'}}^{t_{i'}} + P_{c_{i'}}), \forall i' < i, c_{i'} = c_i \\
& t_i \geq (1 - y_i)(1 - y_{i'})(t_{i'} + \frac{s_{i'}}{B}), \forall i' < i \\
& X_j^t, y_i \in \{0, 1\}, \forall i, j, t
\end{aligned}
$$

where y_i is an indicator to show whether the image I_i is offloaded or processed locally. If $y_i = 1$, it will be processed locally, and it will be offloaded to the edge server if $y_i = 0$. This problem is NP-Hard and thus we design heuristic-based algorithms to solve it.

V. Resource-Aware Scheduling Algorithms

In this section, we first propose a naive algorithm to optimize the completion time, and then propose an advanced algorithm to further reduce the completion time.

A. The Naive Algorithm

For each CNN model, there is a job queue, and jobs in the queue are images to be processed by the CNN model. Since there is limited memory space, some CNN models may not be in the memory and must be loaded into the memory. Therefore, the time for processing all jobs in the j^{th} job queue locally can be computed as:

$$T_j^{loc} = \begin{cases} \sum_{I_i \in U_j} P_j & \text{The } j^{th} \text{ model is in the memory.} \\ L_j + \sum_{I_i \in U_j} P_j & \text{otherwise.} \end{cases}$$

where U_j is the set of images needed to be processed by the j^{th} CNN model. We can also choose to offload all jobs in the j^{th} job queue to the edge server, and the offloading time is $T_j^{off} = \frac{\sum_{I_i \in U_j} s_i}{B}$.

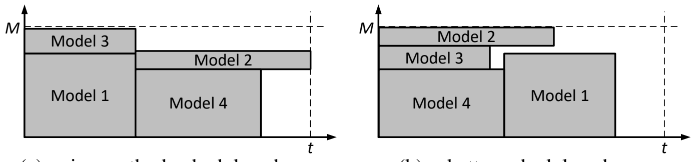
The running time difference between offloading and local processing for the j^{th} job queue is denoted as $\mathscr{D}_j$, where $\mathscr{D}_j = T_j^{loc} - T_j^{off}$. It represents the amount of time to be saved by offloading the jobs in the j^{th} job queue to the server. Therefore, jobs in the job queue that has the largest $\mathscr{D}_j$ should be offloaded, and jobs in the job queue which has the smallest $\mathscr{D}_j$ should be processed locally. Our naive algorithm can be summarized as follow:

1) For the jobs to be processed, compute the running time difference $\mathscr{D}_j$ for the j^{th} job queue, and sort the job queues in ascending order based on $\mathscr{D}_j$.
2) Pick the job queue with the largest $\mathscr{D}_j$ and offload the jobs in this job queue.
3) Meanwhile, starting from the first job queue, load the corresponding CNN models into the memory in ascending order of $\mathscr{D}_j$ until there is not enough memory space for loading another one. The jobs in these job queues are processed locally.
4) Repeat the above steps when a job queue becomes empty. The algorithm terminates when there is no job to be processed.

The Naive algorithm has a low time complexity of $O(n)$, where n is the total number of jobs for all the job queues. Therefore, it can be executed very efficiently. It is simple and can be easily implemented.

B. Our Resource-Aware Scheduling Algorithm

1) Motivation: There are some weaknesses in the Naive algorithm. First, it only considers the running time difference between offloading and local processing for the whole queue, but ignores the data size difference of different images. For the same job queue, offloading the processing of an image with smaller data size can save more time compared to offloading the larger ones. This is because the processing time does not depend on the image size, and all images are scaled to the same resolution to run CNN models. At the same time, it takes less time to offload smaller images than larger ones. Therefore, it is better to offload small images from different queues, instead of offloading images in a specific job queue.



(a) naive method schedule scheme (b) a better schedule scheme

Fig. 4: Suppose the time difference for each job queue is $\mathscr{D}_i (1 \le i \le 4)$ and $\mathscr{D}_1 < \mathscr{D}_3 < \mathscr{D}_2 < \mathscr{D}_4$. Therefore, the naive algorithm first loads Model 1 and Model 3 into the memory and processes the corresponding jobs locally. It fails to find a better solution.

Second, the Naive algorithm may miss some better solutions. Consider the example shown in Fig. 4. In the figure, x axis shows time, and y axis shows the memory usage where M is the maximum available memory. The rectangle represents the job queue of each CNN model. The width of the rectangle stands for the local processing time for the corresponding job queue, while the height represents the runtime memory requirement. Since the Naive algorithm only considers the time difference between offloading and local processing, it generates a schedule shown in Fig. 4(a). However, a better schedule with less completion time can be found in Fig. 4(b).

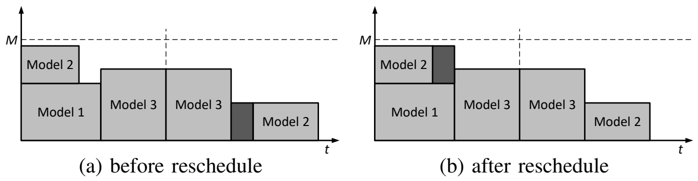
2) Scheduling Image processing within a Time Slot: In the Advanced algorithm, time is divided into fixed time slots, and we first assign jobs to different time slots based on their arrival time. For each time slot, we try to minimize the completion time for processing all images. We first discuss how to optimize the completion time within a time slot, and then discuss how to further reduce the completion time by rescheduling jobs among different time slots.

During scheduling, the Advanced algorithm tries to determine which images should be offloaded for each job queue and optimize the completion time for the local processed jobs.

A brute force method is to list all the possible schedules (i.e., what images for each job queue should be offloaded), compute the running time for each schedule and find out the optimal one among them. For each job queue, time can be saved by offloading the smaller images first. Therefore, the advanced algorithm sorts the images in the ascending order based on the data size in each job queue. For each job queue, it always chooses the images from the head of the queue to be offloaded. Therefore, we have $\prod_j (|U_j|+1)$ different schedule decisions using the brute force method. When the number of images is large, it is impossible to find the optimal.

To improve the search efficiency, we adopt the idea of dynamic programming. We choose a parameter θ to limit the total number of decisions. For each job queue j, the jobs are divided into θ groups. For instance, the first $\frac{|U_j|}{\theta}$ images are put into the first group, and the next $\frac{|U_j|}{\theta}$ images are put into the second group, and so on. All jobs belong to the same group can either be offloaded to the server or run locally. In this way, the number of choices is significantly reduced.

For each possible schedule, Algorithm 1 is used to find out



(a) before reschedule (b) after reschedule

Fig. 5: An example for rescheduling.

the optimal schedule for the jobs that need to be run locally.

Algorithm 1: Local Scheduling within a Time Slot

Data: job queues, the set of CNN models in memory (V)
Result: the local processing time (T which is initialized to be 0), and the loading sequence of these CNN models (S which is initialized to be empty)

```
1  Function Main:
2      Append $V$ to $S$
3      while there exists images to be processed do
4          $V \leftarrow$ FindNextCombination ($V$)
5          Append $V$ to $S$
6          for job queue $j \in V$ do
7              compute $T_j \leftarrow X_j^T L_j + \sum_{I_i \in U_j} P_j$
8          $T \leftarrow T + \min_j T_j$
9          UpdateJobQueues ($T$)
10     return $T, S$
11 Function FindNextCombination($V$):
12     for each job queue $j$ do
13         if $U_j = \emptyset$ then
14             for each valid combination $V'$ do
15                 $V' \leftarrow V' - \{j\}$
16     for each valid combination $V'$ do
17         if $|V'| > |V|$ then
18             $V \leftarrow V'$
19     return $V$
20 Function UpdateJobQueues($T$):
21     for each job queue $j$ do
22         for $I_i \in U_j$ do
23             if $t_i + P_j + L_j X_j^{t_i} \le T$ then
24                 $U_j \leftarrow U_j - \{I_i\}$
```

In Algorithm 1, different job queues need to be processed by different CNN models. Since each CNN model can be in the memory or not, there are many possible combinations. However, due to memory constraint, not all CNN models can be in the memory, and then not all combinations are valid. Then, a valid combination, denoted as V, is a set of CNN models which can be in the memory at the same time. Given the jobs that need to be processed locally, the algorithm finds a loading sequence of CNN models to minimize the local processing time.

3) Rescheduling Across Time Slots: Through the above discussion, we obtain a schedule within each time slot. However, such schedule may not be the best solution across time slots. An intuitive solution is to run all of the scheduling decisions for all time slots one by one, but this method cannot fully utilize the device memory to run jobs locally.

Consider the example shown in Fig. 5, where the notations are similar to that in Fig. 4. Here we have time slots, represented by the vertical dashed line. As shown in Fig. 5(a), for time lot 1, model-2 does not process any image after it finishes its jobs. However, model-2 still occupies the memory and model-3 cannot be loaded before model-1 finishes its jobs in time slot 1. As can be seen, in time slot 1, model-2 has been idle for some time, although it should run the jobs represented with dark color. As shown in Fig. 5(b), with rescheduling across time slots, the jobs represented with dark color is rescheduled from time slot 2 (shown in Fig. 5(a)) to time slot 1. As a result, the processing completion time is reduced.

Algorithm 2: Reschedule across time slots

Data: The schedule for the time slot.
Result: new schedule

```
1  for each model $j$ do
2      Let $t \leftarrow T_j^{In} + \sum_{I_i \in U_j} P_j + L_j$
3      while $T_j^{Out} - t > P_j$ do
4          for $U_j^{'}$ in each of the following time slot do
5              $a \leftarrow \min_{I_i \in U_j^{'}} a_i$
6              if $a \neq \emptyset$ then
7                  break
8          $t \leftarrow \max(a, t)$
9          if $t + P_j \le T_j^{Out}$ then
10             Reschedule the image to this time slot
11             $t \leftarrow t + P_j$
12         else
13             break
14 return new schedule
```

To reschedule jobs across time slots, we first need to find out the time period when a CNN model is idle in a time slot. For a CNN model j in a time slot, let T_j^{In} denote the time it is loaded into the memory, and let T_j^{Out} denote the time that the model is switched out. The idle period for this model is: $(T_j^{In} + \sum_{I_i \in U_j} P_j + L_j, T_j^{Out})$. We can make use of this idle time period and ask model j to process images in the following time slots. The rescheduling procedure is summarized in Algorithm 2.

If some jobs at a time slot are rescheduled to another time slot, their corresponding CNN models will have more idle time. The local processing time for the time slot will not change if we can reschedule enough jobs to it. However, there are some cases that such reschedules cannot be done. For example, we cannot reschedule any jobs to the last time slot, since no jobs will arrive. In such cases, Algorithm 1 will be run again to find a more efficient schedule for this time slot.

VI. Performance Evaluations

In this section, we evaluate the performance of the proposed resource-aware scheduling algorithms: the Naive algorithm and the Advanced algorithm. In the evaluation, we consider two sets of traces, one is generated by us and the other one is based on real world video dataset.

A. Evaluation Setup

In our evaluation, we use a Samsung Galaxy S8 smartphone, and a Dell desktop with Intel i7-3770@3.4GHz CPU and 16GB RAM as the server. We use classic HoG and linear SVM detectors for object detection which can run in real time and achieve high accuracy. We do not use CNN model for object detection since it is too resource-hungry for mobile devices. We choose Caffe [23] as the deep learning framework and choose some of well-known CNN models from Caffe Model Zoo for our tests. Since the original version can only be run on computers, we use the library ported to Android. However, this library does not support for running multiple CNN models simultaneously, thus we modify its code to allow smartphones to load multiple CNN models and execute them at the same time.

The evaluation compares our proposed algorithms with the following three approaches.

- **Offload-Only**: This method always offloads images to the server for processing.
- **Local-Only**: This method processes all input images on the mobile device. Algorithm 1 is used to find out the best schedule.
- **DeepEye**: We implement DeepEye [19] to optimize the video processing with the help of cloud server. DeepEye loads the CNN models into the memory based on the descending order of model sizes until there is not enough memory space for loading another one. The DeepEye does not switch out CNN models from the memory and always offloads the data if the corresponding CNN model is not in the memory.

The input images that need to be processed are randomly generated following uniform distribution. All the input images are assigned a random arrival time ranging from 0 to 100 seconds, and then assigned to a CNN model to be processed. The image size ranges from 40×40 pixels to the maximum input size of its corresponding CNN model. Images that smaller than 40×40 pixels are ignored. All the random values are generated following a uniform distribution. We use workload to represent the number of arriving images per second.

B. Comparisons of Different Approaches in Various Settings

Model	Memory Usage(MB)	Processing Time (s)	Loading (s) Time (s)
Resnet	716	0.95	0.606
CaffeNet	317	2.38	0.36
GoogleNet	472	0.69	0.39
LightCNN	788	0.96	0.2
VGG	563	2.15	0.34
HybridCNN	604	3.91	0.45

TABLE II: The models used in our experiment.

We first evaluate our algorithms under different uplink data rates, and the result is shown in Fig. 6. In this experiment, we use the models described in Table. II.

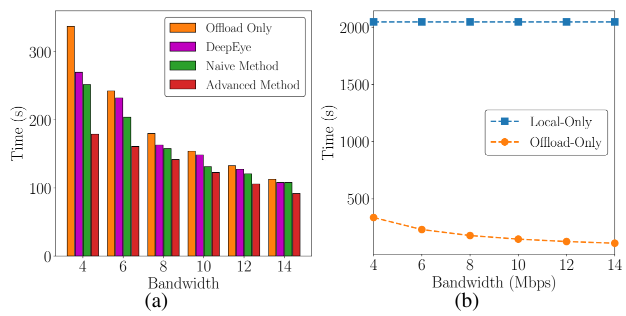


Fig. 6: Performance under various uplink rates.

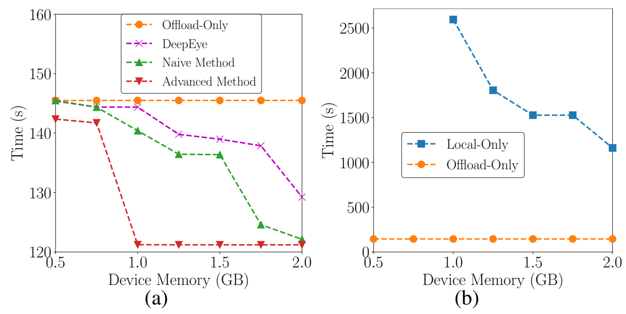
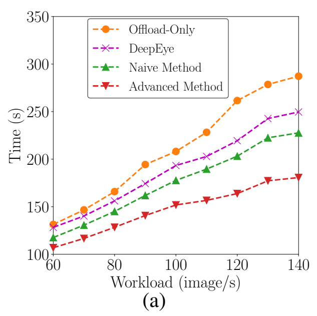


Fig. 7: Performance under various device memory limitations.

We set the workload to be 80 images/s and limit the available memory to be 1 GB. As we can see in Fig. 6(a), when the data rate is small, the advanced algorithm outperforms the DeepEye and naive algorithm and it can save up to 50% of the processing time when the data rate is 4 Mbps. When the data rate becomes larger, DeepEye, the naive algorithm and the advanced algorithm will be similar to the Offload-Only method, since most of the input images will be transmitted to the server for processing. In the experiments, we do not separate the running time of our scheduling algorithms, since the overhead of our scheduling algorithm (0.02s for nave, 0.5s for resource-aware algorithm) is negligible compared to the video processing time (100s level).

In Fig. 7, we compare the five algorithms under different memory limitations. For this test, we still use all the models described in Table. II, and set the workload and the uplink data rate to be 80 images/s and 8 Mbps respectively. Fig. 7(b) compares Offload-Only and Local-Only. When the available memory is less than 700 MB, Local-Only cannot finish all jobs since some of the CNN models require more memory space to run. As the memory size increases, the processing time for the Local-Only method decreases, since it can run more models at the same time. Fig. 7(a) shows the comparison between the naive method and the advanced method.

As can be seen from the figure, some part of the advanced algorithm is flat. This is because the advanced algorithm tries to load as many CNN models as possible. When the memory increases a little bit, it is possible that the advanced algorithm cannot load another model into the memory. Hence, it will still have the same completion time. Since the naive algorithm cannot fully utilize the device memory, it takes 50% more time



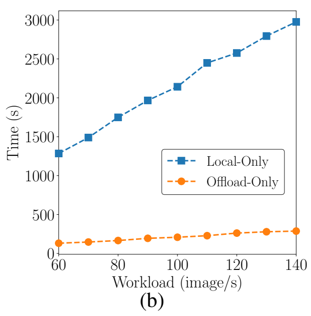
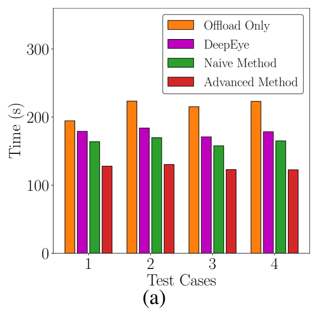


Fig. 8: Performance under various workloads.



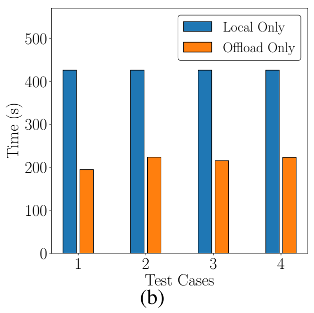


Fig. 9: Performance using various CNN model combinations.

than the advanced one in some cases. When the memory size increases, the performance of both methods becomes close since they can load the same number of CNN models.

In Fig. 8, we perform tests under different workload settings. The bandwidth and the memory limitation are set to be 8 Mbps and 1 GB. Fig. 8(a) compares the performance of Offload-Only, DeepEye, Naive and Advanced. When the workload is low, the processing time of these four algorithms is similar. This is because most devices have enough time to offload most of the images before more images arrive. As the workload increases, not all images can be transmitted in time. At this time, our algorithms can save time by processing some images locally. The advanced algorithm can save more time than DeepEye and the naive one, since it will not only consider the size of the offloaded images, but also utilize the memory space better.

In Fig. 9(a), we evaluate the effects of different combinations of CNN models on our algorithms. There are four test cases, as shown in Table. III, and each test case selects a subset of the CNN models used in previous experiments. The uplink bandwidth and the workload are set to be 8 Mbps and 70 images/s individually. As shown in Fig. 9, the advanced algorithm can save 25%, 30% and 40% of the processing time than the naive algorithm, DeepEye and the Offload-Only method.

Fig. 10 shows an example of model switching on smartphone, and it is taken from the first five time slots in test case 1 shown in Table III. Fig. 10(a) shows the job arrivals for each CNN model, and Fig.10(b) shows the number of local processed jobs. Fig. 10(c) shows which models are in memory at each time slot. At the beginning, due to the variations of image size, not all jobs of a model will be offloaded to the edge server. Resnet and LightCNN are loaded first to process their

Test	Selected Models	Memory
1	All models	1.5 GB
2	Resnet50, caffenet, googlenet, lightcnn, ilsvrc13	1.5 GB
3	caffenet, googlenet, lightcnn, ilsvrc13	1.25 GB
4	Resnet50, googlenet, lightcnn, ilsvrc13	1.25 GB

TABLE III: The test cases used in Fig. 9.

jobs locally, then GoogleNet replaces Resnet and executes its job locally. For time period 2 and 3, since the number of Resnet jobs is small and Resnet has a longer processing time compared with the other two models, all of its jobs are offloaded. At time slot 4, when GoogleNet finishes all its jobs, there are still some jobs of Resnet left, and the Resnet model is loaded to process them locally. At time slot 5, since Resnet is in the memory, it will run its jobs first (a very short time period at the start of time slot 5) and then the GoogleNet model will replace it.

C. *Real World Video Analytics*

Video Index	Text	Person	Vehicle
1	764	1892	0
2	120	1428	48
3	192	1560	6
4	203	13473	337
5	87	2410	101
6	327	1212	0

TABLE IV: The number of detected objects in each video.

Selected Models	Extracted Information
Resnet	text recognition
GoogleNet	vehicle models
Caffenet	age
LightCNN	face features
VGG	gender
HybridCNN	scene detection

TABLE V: CNN models used in real world video analytics.

Besides using generated images to evaluate our algorithms, we also use the real world video data for evaluations. We use videos from the newest dataset in Multiple Object Tracking Benchmark (MOT) [24]. These videos are taken at different places, where some of them capture views on street, and others record scenes in shopping malls. In this experiment, we use different CNN models to extract features from the videos for different objects including people, vehicles and texts. Table IV lists the number of objects detected in the videos. Table. V shows the models used in the experiment.

Most videos in the MOT dataset are captured with a high frame rate, ranging from 25 to 30 fps. It is not necessary to process all frames in the videos since consecutive frames are similar. Thus, before processing videos, we perform a sampling by picking the first frame for every 5 frames. The device memory limitation is set to 1.5 GB and the network bandwidth is set to 8 Mbps.

The results are shown in Fig. 11. Since the Local-Only method significantly underperforms the other four approaches, the figure shows the completion time ratio of different approaches to the Local-Only method. Compared with Offloading-Only, the naive algorithm can reduce the completion time by about 16% on average, whereas the

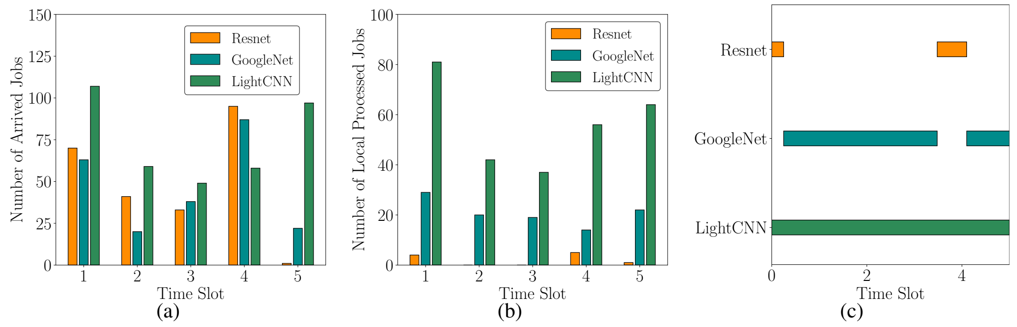


Fig. 10: An example shows model switch in different time slots. (a) the number of jobs that arrives in each time slot, (b) the number of local processing job for each model, and (c) models in memory at each time slot.

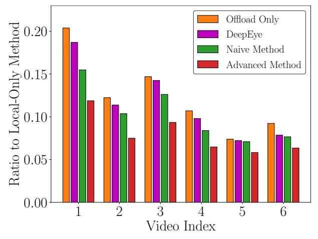


Fig. 11: Performance in the real world video clips.

advanced algorithm can reduce it by about 35%. In addition, the completion time of the advanced algorithm is about 24% shorter than the naive algorithm.

VII. Conclusions

In this paper, we identified the research challenges of running multiple CNN models on mobile devices under resource constraints, and proposed resource aware scheduling algorithms to support multiple CNN models on mobile devices. Our goal is to minimize the completion time of video processing, which is accomplished by first deciding whether an incoming image processing task should be offloaded or executed locally. If a task is decided to be executed locally, its corresponding CNN model must be loaded into the memory, possibly by switching out some other models due to memory constraint. We formulated this scheduling problem as an integer programming problem and proposed two heuristic based algorithms: a naive algorithm which decides whether to offload or run locally based on their completion time difference, and an advanced algorithm which addresses some weaknesses of the naive algorithm to further reduce the completion time. We have implemented the proposed scheduling algorithms on Android-based smartphones and have demonstrated its effectiveness through extensive evaluations.

Acknowledgement

This work was supported in part by the National Science Foundation (NSF) under grant CNS-1815465.

References

[1] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Deep Features for Text Spotting," *ECCV*, 2014.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *IEEE CVPR*, 2016.

[3] ——, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *IEEE ICCV*, 2015.

[4] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning Deep Features for Liscriminative Localization," *IEEE CVPR*, 2016.

[5] O. Parkhi, A. Vedaldi, and A. Zisserman, "Deep Face Recognition," *British Machine Vision Conference*, 2015.

[6] L. Dubourg, A. R. Silva, C. Fitamen, C. J. Moulin, and C. Souchay, "SenseCam: A New Tool for Memory Rehabilitation?" *Elsevier Revue Neurologique*, 2016.

[7] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," *ACM Mobisys*, 2010.

[8] Z. Lu, K. Chan, and T. Porta, "A Computing Platform for Video Crowdprocessing Using Deep Learning," *IEEE INFOCOM*, 2018.

[9] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge," *IEEE INFOCOM*, 2019.

[10] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics," *IEEE INFOCOM*, 2018.

[11] 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," *ACM Mobisys*, 2016.

[12] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," *IEEE ICPR*, 2016.

[13] T. Tan and G. Cao, "Efficient Execution of Deep Neural Networks on Mobile Devices with NPU," *ACM IPSN*, 2021.

[14] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable Adaptation of Video Analytics," *ACM SIGCOMM*, 2018.

[15] 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," *ACM Mobisys*, 2018.

[16] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-efficient computation offloading in cellular networks," *IEEE ICNP*, 2015.

[17] Y. Geng, Y. Yang, and G. Cao, "Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices," *IEEE INFOCOM*, 2018.

[18] T. Tan and G. Cao, "FastVA: Deep Learning Video Analytics Through Edge Processing and NPU in Mobile," *IEEE INFOCOM*, 2020.

[19] A. Mathur, N. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware," *ACM Mobisys*, 2017.

[20] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, and A. Berg, "SSD: Single Shot Multibox Detector," *ECCV*, 2016.

[21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-time Object Detection," *IEEE CVPR*, 2016.

[22] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning Deep Features for Scene Recognition Using Places Database," *NIPS*, 2014.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *ACM International Conference on Multimedia*, 2014.

[24] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler, "MOT16: A Benchmark for Multi-Object Tracking," *arXiv:1603.00831 [cs]*, Mar. 2016, arXiv: 1603.00831.