A Computing Platform for Video Crowdprocessing Using Deep Learning

Zongqing Lu[†], Kevin S. Chan[‡], and Thomas La Porta[§]
[†]Peking University, [‡]Army Research Laboratory, [§]Pennsylvania State University
[†]zongqing.lu@pku.edu.cn, [‡]kevin.s.chan.civ@mail.mil, [§]tlp@cse.psu.edu

***Abstract*—Mobile devices such as smartphones are enabling users to generate and share videos with increasing rates. In some cases, these videos may contain valuable information, which can be exploited for a variety of purposes. However, instead of centrally collecting and processing videos for information retrieval, we consider crowdprocessing videos, where each mobile device locally processes stored videos. While the computational capability of mobile devices continues to improve, processing videos using deep learning, *i.e.*, convolutional neural networks, is still a demanding task for mobile devices. To this end, we design and build *CrowdVision*, a computing platform that enables mobile devices to crowdprocess videos using deep learning in a distributed and energy-efficient manner leveraging cloud offload. CrowdVision can quickly and efficiently process videos with offload under various settings and different network connections and greatly outperform the existing computation offload framework (*e.g.*, with a 2× speed-up). In doing so CrowdVision tackles several challenges: (*i*) how to exploit the characteristics of the computing of deep learning for video processing; (*ii*) how to parallelize processing and offloading for acceleration; and (*iii*) how to optimize both time and energy at runtime by just determining the right moments to offload.**

I. Introduction

Mobile devices such as smartphones are enabling users to generate and share videos with increasing rates. In some cases, these videos may contain valuable information, which can be exploited for a variety of purposes.

In this paper, we consider a video classification problem where a *task issuer* asks the crowd to identify relevant videos about a specific object or target. This requires object detection to be performed on videos to detect these objects of interest within the frames of the videos. The limitations of mobile devices for these types of applications are well known. The most obvious challenge is the computational requirement. Although it continues to improve, video processing using deep learning, *i.e.*, Convolutional Neural Networks (CNNs), is still a demanding task for mobile devices [11].

We anticipate that video processing can be performed within a network of mobile devices and a powerful cloud environment. Individuals have the option to process the videos locally or offload them to a highly capable computing unit in the cloud. Coupled with the computation limitation is the cost to transmit, whether it be via WiFi or cellular (4G LTE). Transmitting videos to the cloud over wireless links consumes considerable energy. Additionally, when using cellular networks, there may be data budgets that limit data transfer without incurring extra expenses. As a result, we consider the coupled problem of constrained resources of computing, data transmission, and battery.

This work was supported in part by Network Science CTA under grant W911NF-09-2-0053 and Hikvision.

Due to these limitations, users may be reluctant to participate in such requested video processing tasks. However, users could be motivated by various incentive mechanisms, such as [15], which are not the focus of this paper. For this paper, we consider a variation on a crowdsensing-type scenario that we call *crowdprocessing*. Instead of users collecting and sharing data to solve a larger problem, we explore the computing capability of mobile devices and allow individuals to process some (or all) of the data rather than having some centralized entity process everything.

We design and build CrowdVision, a computing platform that enables mobile devices to crowdprocess videos using deep learning in a distributed and energy-efficient manner leveraging cloud offload. In doing so we tackle several challenges. First, to design a computing platform specifically for video processing using deep learning, we should take into account the characteristics of the computing of deep learning. By deploying and measuring the computing of CNNs on mobile devices, we identify *batch processing*, which makes deep learning different from common computational tasks and can be exploited for computing acceleration. Second, as frames extracted from a video arrive at a certain rate, to take advantage of the batch processing towards optimizing the processing time, we need to determine when to perform each batch with how many frames. However, the problem turns out to be a NP-hard problem. By carefully balancing the waiting time for frames to be available and the processing time incurred by each additional processing batch, we are able to determine the batch processing to optimize the processing parallelized by offloading. Third, when the data rate for offloading varies widely, it is hard for mobile devices to acknowledge when offload benefits. By correlating signal strength, data rate, and power based on offloading attempts, we are able to sense and seize the right moments when offloading benefits both processing time and energy at runtime.

We envision CrowdVision to be useful for a variety of applications that require content information of videos from the crowd. One common use case is emergency response situations, where municipal agencies ask the public to assist in identifying terrorists or criminals through scanning of their videos, as the FBI did after the Boston Marathon bombing. CrowdVision could handle this request in a smart and automatic way; *i.e.*, it first filters videos based on location and timestamp, and then collectively performs deep learning to find the object of interest.

Contributions: (*i*) We measure and characterize the processing time and resource usage for each component of video processing using deep learning (§III). (*ii*) We design a split-shift algorithm that parallelizes frame offload and local detection to optimize the processing time by taking advantage of batch processing (§IV). (*iii*) We design an adaptive algorithm featured with a backoff mechanism, which determines the offloading at runtime towards optimizing both completion time and energy (§V). (*iv*) We implement CrowdVision on Android with a GPU-enabled server for offload (§VI). (*v*) We show experimentally that CrowdVision greatly outperforms the existing computation offload framework (*e.g.*, with a 2× speed-up) and improves speed and energy usage of video crowdprocessing, and the split-shift algorithm closely approximates the optimum (§VII).

II. Related Work

There are several crowdsensing frameworks similar to CrowdVision. Medusa [12] is a platform that performs crowdsensing tasks through the transfer and processing of media (text, images, videos). Pickle [10] is a crowdsensing application to conduct collaborative learning. GigaSight [13] is a cloud architecture for continuous collection of crowd-sourced video from mobile devices. In contrast to these frameworks, CrowdVision is a computing platform rather than the system or platform that motivates users and collects sensed data.

There is a large body of work that studies computation offload for mobile devices. These can be summarily classified into two categories: building general models for computation offload such as [2] and computation offload based applications such as [4]. However, due to the characteristic of the computing of deep learning for video processing (*i.e.*, batch processing), *none* of these works can be directly and effectively applied to our application and offloading problem.

Optimizing the computing of CNNs on mobile devices has been recently investigated by compressing parameters of CNNs [14], by distributing computation to heterogeneous processors on-board [5], and by trading off accuracy and resource usage [6]. However, unlike these works, we are building a computing platform for video crowdprocessing, which can easily incorporate these optimization technologies whenever they are available for off-the-shelf mobile devices.

III. Overview

CrowdVision is a distributed computing platform that enables mobile devices to participate in video crowdprocessing. In this environment, a task issuer initiates a query by which mobile devices are requested to identify some object of interest within locally stored videos. We consider a crowdprocessing approach to perform object detection/classification of videos. This task includes filtering of videos based on metadata, and then processing the videos to perform object detection. Caffe [1], a deep learning framework, is currently employed to perform object detection on frames extracted from videos.

The details of the query can be of varying specificity (*e.g.*, "find people wearing red hats walking a dog in the park on Tuesday evening"). These details may allow for the individuals to filter out irrelevant videos based on location or timestamp. At the expense of some energy usage, the user can offload the videos to the cloud, where high performance computing can quickly process the videos without energy usage concerns. Alternatively, the user can process some of the frames locally and offload others. Once these videos are processed using deep learning either locally on the mobile device or remotely on the cloud, the task issuer will receive information about the videos related to the query. For frames that are processed on the mobile devices, the user will forward either the tags, the frames of interest, or the entire video to the cloud. Participation in the task may reveal personal information and intrude on users' privacy, but users are allowed to filter out their personal videos. More sophisticated and rigorous treatment of security and privacy control is left as future work.

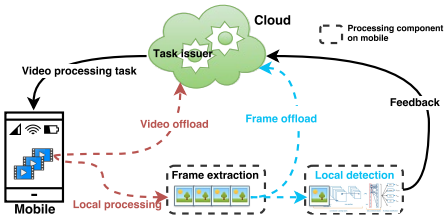
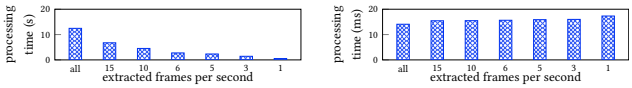


Fig. 1: Overview of CrowdVision.

The overview of CrowdVision is depicted in Fig. 1. A mobile device has several options to perform object detection on each related video. Performing object detection in a video entails two main steps. First, video frames must be extracted from the video and turned into images. Second, object detection is performed on the images. These two functions may be performed *locally* on a mobile device, or *in the cloud*, or *distributed between the two systems*. As illustrated in Fig. 1, by considering several input parameters and constraints, *e.g.*, network connections, battery life, and data budget, a mobile device can perform (*i*) *video offload* by sending the whole video to the cloud. Alternatively, the user may perform frame extraction locally and then offload specific frames to the cloud. In this case, it needs to determine whether each frame is processed by (*ii*) *frame offload* in which the frame is sent to the cloud for detection or (*iii*) *local detection* where the frame is detected on the mobile device.

In CrowdVision, we consider both the task issuer's query requirements and the resource usages of mobile devices. From the perspective of the task issuer, the quality of CrowdVision's response to the query can be evaluated by two metrics. First is the *timeliness* of the response, measuring the time required to process all related videos on each mobile device. Second, the *accuracy* of the response is measured by the frames identified as containing the event that actually contain the event, which is determined by the CNN model. In this work, we consider timeliness. From the users' perspective, its primary criteria is the resource efficiency. Moreover, video processing using deep learning on mobile devices is still limited and resource intensive. Therefore, the cloud is provided by task issuers to assist in video processing.

When mobile devices perform video processing with of-



(a) total processing time (b) processing time per frame

Fig. 2: Processing time of different extraction rates on a 30-second 1080P video using DSP on smartphone.

floading, they may be connected to the cloud via WiFi or cellular networks. In these two circumstances, users may have different priorities in terms of budgeting resources. When using WiFi, users may not care about the resource usage as long as video processing does not affect the normal use of their mobile devices; or they may only care about battery life when battery recharging is not accessible. When using cellular networks, battery life and data usage may become the primary concerns since it is usually inconvenient to recharge mobile device when connected cellular networks and cellular data plans may be limited. Therefore, the design of CrowdVision takes both of these into consideration.

Through the characterization of processing time, energy, and cellular data usage for video processing, we design CrowdVision. By considering inputs from the processing task, constraints of users, and various network scenarios, we optimize cloud-assisted video crowdprocessing as a function of these design parameters. CrowdVision determines how to quickly and efficiently process each video and takes advantage of some actions that can be done in parallel or pipelined. In the following, we describe each of these processing components.

A. Frame Extraction

Frame extraction is used to take individual video frames and transform them into images upon which object detection may be performed. To target objects with different dynamics within the video, the task issuer may request a different frame extraction rate. For example, for an object moving at a high speed like a car, the rate should be high enough to not miss the object. For a slowly moving object like a pedestrian, a lower frame extraction rate can be used, which eases the computing burden. The setting of the frame extraction rate for object detection is important but it is not the focus of this paper. CrowdVision takes frame extraction rate as a parameter defined by the task, denoted as e_r frame per second (fps).

Fig. 2a and 2b, respectively, illustrate the total processing time and the processing time per frame for frame extraction using a hardware codec (DSP) with different extraction rates on a 30-second 1080p video (30fps) on a Galaxy S5 (unless stated otherwise all measurements are performed on the Galaxy S5). In Fig. 2a, the total processing time for extracting all the frames takes about 13 seconds, and it decreases as the extraction rate reduces. The processing time per frame, as illustrated in Fig. 2b, slightly and linearly increases as the extraction rate decreases, and the average is about 16 ms. Moreover, as shown in Table I, the power of frame extraction is about 1W, and the CPU usage is only about 4%.

B. Detection

Detection is the process of determining if the object of interest is in a frame. For detection, we currently use AlexNet

	Power	CPU usage
Frame Extraction	1178 mW	4%
Object detection	2191 mW	25%

TABLE I: Power and CPU usage of frame extraction and object detection on smartphone

[9], a 8-layer CNN, on Caffe to perform classification, but we do not have any restriction on CNN models. Although Caffe can be accelerated by CUDA-enabled GPUs, it is currently not supported by GPUs on off-the-shelf mobile devices. Caffe in both the cloud and mobile devices is the same, but the cloud is equipped with powerful CUDA-enabled GPUs and can perform object detection hundreds of times faster than mobile devices. We find that the processing time of detection is affected by the batch size, *e.g.*, processing two frames in a batch takes less time than that of two frames that are detected individually. Fig. 3 shows the measured processing time of detection performed individually and in a batch. The processing time grows linearly with the increase of the number of frames for both cases. However, batch processing performs much better, where the intercept (α) is about 240ms and the slope (β) is 400ms. The difference grows with the increase of the number of frames. For detection, it is better to put more frames in a batch to reduce processing time. However, the system must wait longer to get the extracted frames from the video. Therefore, it is difficult to determine the best batch size.

This characteristic of batch processing commonly exists in the computing of CNNs on both CPUs and GPUs, and it also drives the design of CrowdVision's modules of local detection and frame offload. Although CrowdVision is designed based on the detection using mobile CPUs, it can be easily adapted to mobile GPUs when they are available for the acceleration of the computing of deep learning on off-the-shelf mobile devices, because this just requires a change of system parameters. Moreover, despite the fact that mobile GPUs can perform several times faster than CPUs, offloading may still be needed since CNN models go deeper and computing becomes much more difficult (*e.g.*, the state-of-the-art CNN, ResNet [7], has more than a thousand layers).

Table I gives the measured power and CPU usage of performing detection on the smartphone. The power is 2191mW, while the CPU usage is 25% (occupied one of four cores). Although object detection requires considerable computation, together with frame extraction, it will not affect the normal operations of mobile devices, and thus we also consider performing video processing on mobile devices locally.

C. Offloading

Mobile devices can choose between video offload and local processing as depicted in Fig. 1. When choosing local

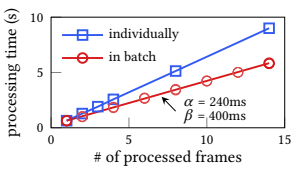


Fig. 3: Processing time of detection performed individually or in batch on smartphone.

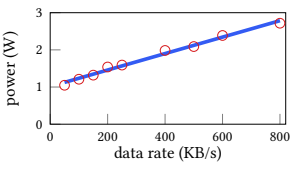


Fig. 4: Power of LTE in terms of different uplink rates on smartphone.

processing, they then decide between frame offload and local detection. These decisions are all affected by network conditions that lead to divergent throughput and power. Moreover, mobile devices can be connected to the cloud via either WiFi or cellular. Each may have different characteristics, and users may also have different concerns in each scenario. Therefore, we separate these two scenarios throughout the design of CrowdVision for ease of presentation. However, it does not mean that the solution proposed for one scenario works exclusively in that scenario.

IV. Processing under WiFi

When mobile devices are connected to the cloud with WiFi, energy usage may or may not be the concern. Therefore, we first study the optimization of the processing time.

A. Optimizing Completion Time

When a mobile device receives a task, it first parses the task to find the related videos based on metadata, such as location information, or timestamp. The completion time is the time spent processing these related videos. We say a video is processed when the video is offloaded to the cloud, or frames are extracted and processed by some combination of the cloud and local device. Multiple videos are processed in serial; *i.e.*, we do not consider parallel video offloading and local processing, since it is not resource-efficient. For example, when local processing is better in terms of both performance and resource usage, video offloading should not be performed at all considering constrained resources on mobile devices. As videos are processed in serial, optimizing the completion time of all related videos on a mobile device is equal to minimizing the completion time of each video. Therefore, for each video, CrowdVision first estimates and compares the completion time of video offload and local processing.

Mobile devices are usually connected to WiFi when users are at home or at work. In such environments, the channel conditions commonly vary slightly. Therefore, similar to MAUI [2], for the processing under WiFi, the data rate between a mobile device and cloud can be considered to be stable during a short period time. Note that the environment with a highly dynamic WiFi data rate can be handled by the adaptive algorithm discussed in §V. Like MAUI, mobile devices probe the data rate by sending 10KB data to the cloud periodically. Let r denote the data rate.

For each video, the processing time of video offload can be easily estimated. However, for local processing, we have to consider the time spent on extracting a frame, offloading a frame, and detecting a frame, and then choose between frame offload and local detection wisely for each frame to obtain minimal completion time of local processing.

Processing time on frame extraction. As shown in Fig. 2b, the processing time of extracting a frame linearly increases as extraction rate decreases. Given an extraction rate defined by the task, we can easily obtain the processing time to extract a frame from a specific video. Note that videos with different resolutions (e.g., 720p, 1080p, 4K) have varied processing times due to different decoding workload. However, the processing time on the frames from videos with the same specifications varies only slightly based on our experiments.

Processing time on local detection. As illustrated in Fig. 3, the processing time of frame detection can be calculated as $\alpha + \beta x$, where x is the number of frames included in a processing batch. However, since extracted frames do not become available at the same time, and batch processing requires all the frames to be processed to be available before processing, the optimized processing time of multiple extracted frames cannot be simply computed as above. This problem turns out to be non-trivial.

Assume there will be totally n frames extracted from a video. Each frame will be available at a time interval γ (*i.e.*, the time to extract a frame). To minimize the processing time, we need to optimally determine how to process these frames, *i.e.*, how many processing batches are needed and how many frames each batch should process.

To mathematically formulate the problem, let us assume the number of batches is also n (n can be seen as the maximum number of batches needed to process the frames) and let y_i denote the number of frames for batch $i \in [1,\, n]$, where $y_i \geq 0$ ($y_i = 0$ means batch i does not process any frame). Let x_i denote the time interval between the start times of batch i and $i+1$, x_0 denote the waiting time before processing the first batch, x_n denote the processing time of the last batch. Note that if y_i and y_{i+1} are both zero, x_i is also zero. Then, our problem can be formulated as an Integer Linear Programming (ILP) problem as following:

$$\min \quad \sum_{i=0}^{n} x_i \tag{1}$$

$$\textit{s.t.} \quad \sum_{i=1}^{n} y_i = n, \tag{2}$$

$$\sum_{i=0}^{k} x_i \geq \sum_{i=1}^{k+1} \gamma y_i, \qquad k = 0, \ldots, n-1 \tag{3}$$

$$x_i + b_i \alpha \geq \alpha + \beta y_i, \qquad i = 1, \ldots, n \tag{4}$$

$$n(1 - b_i) \geq y_i \geq 0, \qquad i = 1, \ldots, n \tag{5}$$

$$b_i = \{0,\, 1\}, \qquad i = 1, \ldots, n. \tag{6}$$

Constraint (2) regulates that all frames are processed. Constraint (3) makes sure that the number of frames to be processed by a batch are available before the start of the batch. Moreover, the previous batch must have been completed already before processing a batch, which can be represented as "$x_i \geq \alpha + \beta y_i$ if $y_i > 0$, otherwise $x_i \geq 0$." Constraints (4)(5)(6) are the workaround for this. If $y_i > 0$, b_i is zero according to constraint (5). If $y_i = 0$, constraint (4) is equal to $x_i + b_i \alpha \geq \alpha$. As our problem is a minimization problem, b will be equal to one. Since the problem (1) is ILP (NP-hard), the optimal solution of minimizing the processing time on detection costs too much, even for a small n. For example, for a instance of $n = 100$ with α, β, and γ obtained from Fig. 2 and 3, GLPK takes 217 seconds to solve it optimally on a 16-core workstation.

Processing time on frame offload. Besides detecting frames locally, mobile devices can also choose to offload frames to

the cloud to accelerate the processing. Let δ denote the time a mobile device spends offloading a frame, where $\delta = d_f/r$ and d_f is the data size of an extracted frame. Note that the extracted frame may be resized to feed different CNN models and thus d_f is not a fixed size. When offloading a frame takes less time than extracting a frame (*i.e.*, $\delta \leq \gamma$), the completion time of local processing is about γn, which is the processing time of frame extraction. However, when $\delta > \gamma$ (the most common case), there will be frame backlog and local detection is needed. However, as discussed above, minimizing the processing time of local detection is already an NP-hard problem. The problem that considers both frame offloading and local detection to minimize the processing time of local processing is even more difficult to solve. Therefore, we propose a *split-shift* algorithm to solve the problem.

B. Split-Shift Algorithm

For each extracted frame, we have two options: frame offload and local detection, which are two processes working in parallel to reduce the completion time of a video. Intuitively, the offloading process should keep sending extracted frames to the cloud. For the detection process, it is better to reduce the number of processing batches (*i.e.*, increase batch sizes), since each additional batch incurs more processing time. However, as the batch processing requires the input frames be available before the processing starts, it is better to not wait too long for the extracted frames. Based on this intuition, we design the *split-shift* algorithm.

If only frame offload is deployed, the completion time is δn (accurately it is $\delta n + \gamma$, but γ is small and cannot be reduced and thus we consider δn for simplicity). We treat the detection process as a helper to reduce δn. The main idea is to shift the workload from the offloading process to the detection process to balance the completion times of these two processes by determining the number of processing batches and the number of frames to be processed in each batch.

First, let us assume that all frames are available at the beginning. Then, let n_p^* denote the number of frames to be detected locally that minimizes the completion time, and we have

$$\delta(n - n_p^*) = \alpha + \beta n_p^*,$$

which can be employed to approximate the case where $\delta \gg \gamma$ and $\alpha \gg \gamma$. Moreover, n_p^* can be seen as the maximum number of frames to detect for all cases. As frames are extracted at a certain rate, the number of frames for location detection should be less than n_p^*.

Since the offloading process keeps sending frames to the cloud, for the detection process, the extracted frame arrives every $\frac{\delta\gamma}{\delta-\gamma}$, denoted by γ'. Then, frame detection can be determined by the following steps. Let b denote the number of processing batches and initially $b = 1$.

1. First, we compare $n_p^*\gamma'$ and α. If $n_p^*\gamma' \leq \alpha$, then we can calculate n_p by solving
$$\delta(n - n_p) = n_p\gamma' + \alpha + \beta n_p,$$
where $n_p = \lfloor \frac{\delta n - \alpha}{\gamma' + \beta + \delta} \rfloor$. For this case, there is only one processing batch and n_p frames.
2. If $n_p^*\gamma' > \alpha$, which means the waiting time before n_p^* frames are available is more than the processing time for an additional processing batch (*i.e.*, α), it is better to schedule more than one batch. Therefore, we increase b by one. Then, n_p^1, *i.e.*, the number of frames of the first processing batch, will be calculated by
$$n_p^1\gamma' + \alpha + \beta n_p^1 \geq n_p^*\gamma'$$
to guarantee that all other frames (*i.e.*, $n_p^* - n_p^1$) are available for detection after n_p^1 is processed, and $n_p^1 = \lceil \frac{n_p^*\gamma' - \alpha}{\gamma' + \beta} \rceil$.
3. However, there may still be $n_p^1\gamma' > \alpha$. If so, it is better to split the first processing batch into two, similar to the previous step. The *split* process continues until $n_p^1\gamma' \leq \alpha$ and then we have the number of scheduled batches and also the number of frames for each batch.
4. n_p^* is derived based on the assumption stated above. However, the frames to be locally detected must be less than n_p^*. Therefore, we need to rebalance the completion time between these two processes by shifting frames from local detection to frame offloading. To do so, first we calculate n_p by
$$\delta(n - n_p) = n_p^1\gamma' + \alpha b + \beta n_p.$$
If $\sum_{i=1}^{b} n_p^i - n_p \geq n_p^b$, decrease b by one and recalculate this equation. The *shift* process is repeated until $\sum_{i=1}^{b} n_p^i - n_p < n_p^b$ and n_p^b is set to $n_p^b + n_p - \sum_{i=1}^{b} n_p^i$. Finally, local detection will process total n_p frames by b batches, and each batch i will process n_p^i frames.

The computational complexity of the algorithm is $O(n)$, which is desirable for mobile devices. Moreover, it is also easy to be implemented on mobile devices; *i.e.*, the offloading process simply keeps sending frames one by one until frame queue is empty, while the detection process initiates batch processing when the number of frames required by each batch are available in the frame queue. Overall, for each video, a mobile device first estimates the completion time of video offload and local processing, respectively, and then chooses the approach that has a better completion time.

Users may also be concerned about energy. The optimization with energy constraint, which can be imposed by the user to govern the energy usage, is also important and left as future work.

V. Processing under Cellular

When mobile devices have only cellular connections, it is better to not offload videos, due to the limitation of cellular uplink speed and limitations on data usage. Since data transmission rates may vary widely over time, *e.g.*, due to movement, the solution for processing under WiFi may not be valid for the cellular scenario.

Obviously, if all frames are detected locally, there is no cellular data usage. However, this may consume significant amounts of energy and severely increase completion time. Although users may be more sensitive about data usage rather than energy, energy usage is a concern. Data usage can be easily controlled, while the energy cost of frame offload varies with data transmission rate and signal strength, and thus it is

hard to tell how much energy will be consumed to offload a frame beforehand.

Therefore, for processing under cellular, we consider the problem of optimizing both completion time and energy consumption with a data usage constraint so that decisions are made while considering tradeoffs between these two objectives. However, due to the variation of cellular data rates, we cannot solve the problem traditionally by Pareto optimal solutions. Thus, we design an adaptive algorithm that makes a decision on each extracted frame (*i.e.*, between frame offload and local detection) towards optimizing both objectives.

To perform video processing under cellular, users need specify a data usage constraint D' between zero and a maximum, which can be easily calculated, *i.e.*, video duration $\times$ frame extraction rate $\times$ frame data size. If $D' = 0$, the problem is trivial and all extracted frames are detected locally. If the user does not care data usage, D' is simply set to the maximum. In the following, we consider the case that $D' > 0$.

A. Estimation of Uplink Rate and Power

Before deciding between detection and offloading for each frame, we need to know the cost in terms of processing time and energy. For frame detection, both processing time and power are stable and can be accurately estimated, although the processing time varies with the number of processing batches. The difficulty lies in estimating these for frame offload. The offloading time and power are related to many factors, such as signal strength, channel conditions and network traffic. However, from measurements on the smartphone, we can see the power of LTE uplink exhibits an approximately linear relationship with data rates as depicted in Fig. 4. Similar results are also found in [8]. Therefore, during a short period time, we can explore the history of previous frame offloads to correlate data rates and power. Moreover, among the factors that affect cellular uplink rates, signal strength, which is mainly impacted by the users' location and movement, can be treated as an indicator of data rate during a short period of time, where the coefficients of the linear relation between data rates and power are steady.

To estimate the uplink data rate and power, first we record the data rate and energy consumption for each offloaded frame. Then, these records can be exploited to derive the linear relation between data rate and power using regression, to maintain an up-to-date correlation estimate. Moreover, we also record the cellular signal strength level during each frame offload. Since generally data rate has a linear relationship with signal strength during short periods of time, we can exploit current signal strength level to roughly estimate current data rate based on the records of previously offloaded frames. Then, the estimated data rate is exploited to gauge energy consumption to offload a frame.

B. Adaptive Algorithm

As aforementioned, we try to optimize completion time and energy consumption together. However, due to the dynamics of cellular uplink data rate and power, we propose a tailored solution for this specific problem rather than a scalar treatment of these two objectives.

For the processing under cellular, we have two options: frame offload and local detection. Frame offload may improve completion time or energy consumption or both, depending on the cellular data rate and power consumed by cellular during offloading. For a number of frames, local detection can be exploited to reduce the completion time by increasing the number of processing batches, although this incurs an additional energy cost. Moreover, local detection usually takes multiple frames as input, and once it starts processing, the frames should not to be offloaded to avoid duplicate processing. Therefore, it is difficult to determine the number of frames included in batch processing, since offloading frames may improve performance during batch processing.

Frame offload may be exploited to reduce both completion time and energy consumption simultaneously; local detection cannot optimize these two objectives together, and hence it may be preferred only when both completion time and energy cannot benefit from frame offload. In addition, we do not parallelize frame offload and local detection, since this always sacrifices one objective for another. Based these considerations, we design an adaptive algorithm for the processing under cellular, towards optimizing both completion time and energy cost with the cellular data usage constraint, which works as follows.

Frames are continuously extracted from a video into a frame queue. When a frame is available, a mobile device will offload a frame to the cloud. If both the offloading time and energy consumption are less than those estimated for local detection, this offloading is called a successful attempt; otherwise, it is referred to as an unsuccessful attempt.

After the first successful attempt, the mobile device will offload another frame. If this is also a successful attempt, then we can derive the linear relationship between data rate and energy consumption, and the relation between signal strength and data rate. For subsequent offloads, the mobile device can estimate the completion time and energy consumption based on current signal strength and the linear functions. If both are less than those estimated for local detection, it will offload another frame.

Whenever an attempt is unsuccessful, or the estimate indicates an unsuccessful attempt will occur, the mobile device will switch to local detection and set a *backoff timer* to ω for frame offload (*i.e.*, frame offload will not be performed during ω). Let $T^p_{\min}(n_r)$ denote the minimal completion time if all the remaining frames n_r are detected locally, n_o denote the number of previously offloaded frames, and n_d denote the maximum number of frames that can be offloaded to the cloud under the data usage constraint D', where $n_d = \lfloor \frac{D'}{d_f} \rfloor$. Then, $\omega = \frac{T^p_{\min}(n_r)}{n_d - n_o} 2^u$, where u is the number of consecutive unsuccessful attempts.

For local detection, each time the mobile device will process as many frames from the frame queue (it may wait for frames to be extracted from a video), which can be completed within

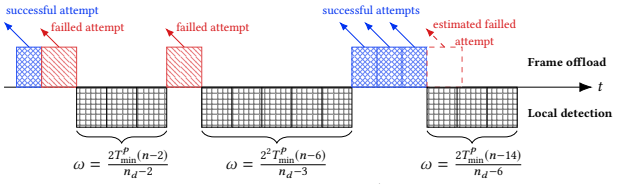


Fig. 5: Illustration of video processing under cellular.

ω. After a timeout of the backoff timer, the mobile device will switch back to frame offload. The process iterates until frame offload reaches the limit n_d or all frames are processed. If limit is reached, then local detection will be the only option to process the remaining frames.

Frame offload and local detection are performed alternately to find the moments when frame offload can improve both energy and completion time, or when local detection should be performed. The backoff timer exponentially increases with the number of consecutive unsuccessful attempts. This is designed to capture different network conditions. When the network condition is constantly poor, offloading is less frequently attempted and the frequency is exponentially reduced. When the network condition varies, offloading is attempted more frequently to seize the moments frame when offload benefits both.

We use Fig. 5 as an example to illustrate how the adaptive algorithm works. Since the first offloading attempt is successful, it offloads the second frame. However, the offloading time is more than detecting a frame locally. Thus, the frame offloading backoff timer is set at $\omega = \frac{2T^p_{\min}(n-2)}{n_d-2}$ and local detection is performed instead during ω. After the timeout, another frame is offloaded, but it fails. Therefore, the mobile device switches to local detection again, and the backoff timer is set to $\frac{2^2T^p_{\min}(n-6)}{n_d-3}$ since there are two consecutive attempt failures. After switching back to frame offload, several successful attempts are made, and hence the mobile device can estimate the data rate and the energy to be consumed based on the signal strength. However, a subsequent estimate indicates that the next attempt will be unsuccessful and hence it performs local detection instead.

Instead of parallelizing frame offload and local detection, we choose to perform them alternatively. Frame offload is employed to optimize both completion time and energy consumption. However, due to the variation of network conditions, frame offload is selected only if it outperforms local detection in terms of both completion time and energy and meets the data usage constraint. The adaptive algorithm is simple and efficient, and it can also be easily implemented.

VI. Implementation

We have implemented CrowdVision on Android and a workstation with a GTX TITAN X GPU as the cloud to assist mobile devices for video crowdprocessing. Tasks are issued from the workstation to mobile devices through Google Cloud Messaging. Messaging between the cloud and mobile devices are implemented using Protocol Buffers. The implementation of CrowdVision on mobile devices consists of three components: core services, monitors, and a GUI.

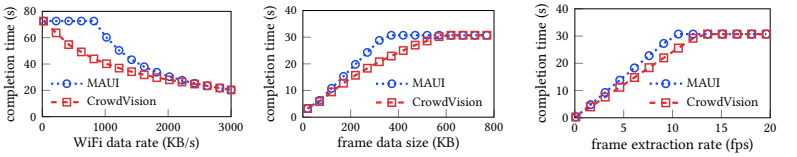
Core Services. Core services contain an executor service, a frame extraction callable, a Caffe callable, an offloading callable, a multithreaded frame queue, and a video database. The video database stores the metadata of local videos such that CrowdVision can quickly screen videos for the processing task. The Multithreaded frame queue stores extracted frames from videos and supports multithreading access. The executor service is able to call any of the callables to perform frame extraction, frame detection, or offloading of a frame or video. The executor service takes inputs from tasks, GUI and monitors, and employs the selected strategy to process each video.
Monitors. There are two monitors to measure network state and battery level, which provide inputs to core services. The network monitor tells core services current network connection with distinct metrics for WiFi or cellular networks. For WiFi, it measures the uplink data rate from a mobile device to cloud using probing data. For cellular, it monitors the signal strength level. The battery monitor measures the energy consumption for each offloaded frame for the cellular case.
GUI. The GUI allows mobile users to configure the energy usage, cellular data usage, and access to videos. Energy control is enabled using WiFi. For cellular, the data usage limits can be specified by users. Additionally, users can select videos to not be processed by CrowdVision for privacy concerns.

VII. Evaluation

In this section, we first compare CrowdVision against alternatives based on empirically gathered measurements to understand when and why CrowdVision outperforms alternatives and then confirm the gains of CrowdVision through experiments on our testbed. We also investigate how the split-shift algorithm approximates the optimum for determining processing batches.

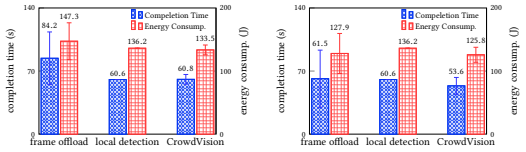
A. Performance under Various Settings

We first use empirically gathered measurements of processing time and energy taken from a Galaxy S5 to investigate the impact of system parameters, such as WiFi/cellular data rate, frame data size and frame extraction rate. Under WiFi, CrowdVision is compared against MAUI [2] (the most popular computation offload system), where MAUI is adapted to optimize the completion time. Under cellular, as MAUI does not adapt to varying data transmission rates, CrowdVision is compared against basic processing options (*i.e.*, solely frame offload and solely local detection). The simulation is carried out on a 1080p 30-second video. Note that the video specification and duration do not affect their relative performance.
WiFi. Fig. 6a illustrates the completion time of MAUI and CrowdVision in terms of WiFi data rate, where the frame data size d_f is 500kB and frame extraction rate r_e is 6fps. When the WiFi data rate is high enough, *e.g.*, 2620KB/s in Fig. 6a, video offload costs the least and both MAUI and CrowdVision choose video offload and hence perform the same. Similarly, when the WiFi data rate is low enough, local detection will be selected by both MAUI and CrowdVision and thus they perform equivalently again, *e.g.*, 20KB/s in Fig. 6a.



(a) $d_f = 500$KB/s, $r_e = 6$fps (b) $r_e = 6$fps, $r = 2000$KB/s (c) $d_f = 500$KB/s, $r = 2000$KB/s

Fig. 6: Performance of CrowdVision and MAUI under WiFi in terms of WiFi data rate, frame data size, and frame extraction rate.



(a) [20, 100, 150, 250, 400KB/s] (b) [20, 100, 150, 500, 600KB/s]

Fig. 7: Performance under different cellular data rates, where $d_f = 100$KB, $r_e = 6$fps.

When between these two rates, CrowdVision outperforms MAUI. This is because the split-shift algorithm employed by CrowdVision can improve completion time by taking advantage of batch processing and by parallelizing frame offload and local detection. The completion time of CrowdVision is as low as 60% of MAUI as depicted in Fig. 6a.

Since CNN models may require different resolutions of images as input, we also investigate the effect of frame data size on the completion time. As illustrated in Fig. 6b, similar to the effect of WiFi data rate, when frame data size is large enough, it is always better to offload videos. When frame data size is small enough, frame offload is the best option. In both regions, MAUI and CrowdVision perform equally. However, between them, CrowdVision always outperforms MAUI as depicted in Fig. 6b due the same reason discussed above. To detect different objects, different frame extraction rates may be defined by the task issuer. Similar to the effect of WiFi data rate and frame data size, CrowdVision outperforms MAUI as shown in Fig. 6c.

Cellular. Based on the measurements of the LTE module on a Galaxy S5 under different uplink rates, as illustrated in Fig. 4, we perform the evaluation over cellular networks.

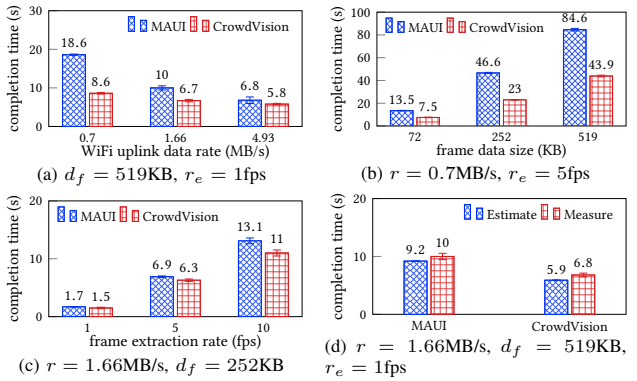
To model the dynamics of cellular data rate, we adopt a Markov chain [3]. Let R denote a vector of transmission rates $R = [r_0, r_1, \ldots, r_l]$, where $r_i < r_{i+1}$. The Markov chain advances at each time unit. If the chain is currently in rate r_i, then it can change to adjacent rate r_{i-1} or r_{i+1}, or remain in r_i, but staying in current rate has a larger probability than changing. Therefore, for a given vector, *e.g.*, of five rates, the transition matrix is defined as: if a rate has two neighboring rates, the probability of staying current rate is 5/9, and the probability of changing to either of two neighboring rates is 2/9; if a rate has only a neighboring rate, the probability of staying current rate and changing to the neighboring rate is 2/3 and 1/3, respectively. In the experiments, we consider two vectors of cellular data rates, which are R_1 = [20, 100, 150, 250, 400] (KB/s) and R_2 = [20, 100, 150, 500, 600] (KB/s), and the time unit of the Markov chain is two seconds. Moreover, we set the data usage constraint of CrowdVision to half of total extracted frames.

Fig. 7a and 7b illustrate their performance under R_1 and R_2, respectively. As depicted in Fig. 7a, CrowdVision outperforms frame offload in terms of both completion time and energy; its performance is similar to local detection. When cellular data rates increase to R_2 in Fig. 7b, frame offload performs better than before as expected. Since cellular data rates do not affect local detection, its performance remains the same. CrowdVision performs the best. It has less completion time than others and slightly less energy consumption than frame offload, with *half* of cellular data usage. CrowdVision outperforms frame offload and local detection under different cellular data rates, because the adaptive algorithm of CrowdVision is designed to adopt different processing options according to real-time cellular data rate and energy to be consumed. Generally, when the cellular data rate is high, it tends to offload frames, otherwise, it is apt to perform local detection. Moreover, the backoff mechanism avoids unnecessary frame offload when the cellular data rate is low.

B. Performance on Testbed

Testbed. We deployed CrowdVision on a Galaxy S5, which can connect to the Internet using either WiFi or cellular. The workstation can be reached by a public IP address. For the experiment using WiFi, we configured a WiFi router with different 802.11 protocols to get different uplink data rates. The experiments under cellular (4G LTE) were carried out at three different locations (*i.e.*, a lab, a lobby, and a restaurant in downtown) to acquire different signal strengths, uplink data rates, and traffic conditions. The performance is evaluated based on the processing of a 1080p 30-second video under different settings in terms of completion time and energy, where energy is measured by the Monsoon power monitor.

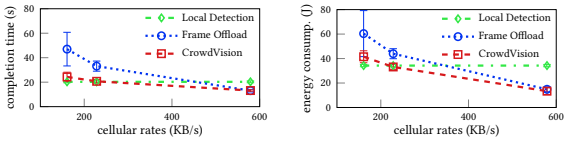
WiFi. Fig. 8 illustrates the completion time of MAUI and CrowdVision. The experiments are conducted using the following parameters: uplink data rates (0.7, 1.66 and 4.93 MB/s), frame data sizes (72, 252, and 519 KB, which are the sizes of JPG images with resolutions 640×360, 1280×720 and 1920×1080, respectively), and frame extraction rates (1, 5, 10 fps). Fig. 8a, 8b and 8c depict the completion time in terms of WiFi uplink data rate, frame data size and frame extraction, respectively. They exhibit the similar pattern as depicted in Fig. 6a, 6b and 6c. CrowdVision outperforms MAUI in all the



(a) $d_f = 519$KB, $r_e = 1$fps (b) $r = 0.7$MB/s, $r_e = 5$fps

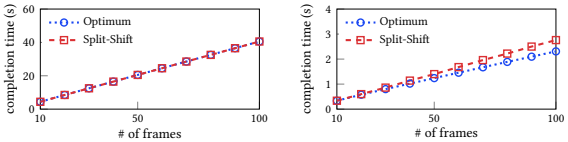
(c) $r = 1.66$MB/s, $d_f = 252$KB (d) $r = 1.66$MB/s, $d_f = 519$KB, $r_e = 1$fps

Fig. 8: System performance of different processing options with various settings and different WiFi data rates.



(a) completion time (b) energy consumption

Fig. 9: System performance of different processing options under different cellular uplink rates, where $d_f = 252$KB, $r_e = 1$fps.



(a) $\alpha = 240$, $\beta = 400$, $\gamma = 16$ (b) $\alpha = 40$, $\beta = 20$, $\gamma = 16$

Fig. 10: Comparison between Optimum and Split-Shift.

settings. For example, when WiFi uplink data rate is 0.7MB/s, the completion time of CrowdVision is less than a half of MAUI (*i.e.*, 2x speed-up). These experiments further confirm the gain of CrowdVision over MAUI. Moreover, as shown in Fig. 8d, the estimated completion time is very close to the measured completion time. Therefore, CrowdVision can reliably make the best decision based on the estimate.

Cellular. We measured the average uplink data rates at the three locations and correlated the performance with these rates, as illustrated in Fig. 9. When the data rate is low, CrowdVision outperforms frame offload in terms of both completion time and energy. Since CrowdVision has to send some frames to acknowledge the current cellular data rate, it incurs slightly longer completion time and uses more energy than local detection. When the data rate increases, CrowdVision and frame offload perform better than before. CrowdVision has similar completion time with local detection but uses less energy. When the data rate further increases, CrowdVision is faster and uses less energy than the others. When the data rate is sufficiently high such that offloading a frame commonly outperforms locally detecting a frame in both completion time and energy cost, CrowdVision tends to offload more frames but avoids the time when the data rate is low. Therefore, CrowdVision adapts to different cellular data rates to reduce completion time and save energy.

C. Performance of Split-Shift

Although the split-shift algorithm is designed to handle the local processing under WiFi, it can also be exploited to solve the ILP problem (1) by assuming $\delta \to +\infty$ (recall δ is the time spent to offload a frame). The ILP problem is meaningful. It can be employed to minimize the processing time of deep learning on frames extracted from videos, no matter the processing is performed on CPUs or GPUs. Therefore, we investigate the performance of the *split-shift* algorithm on (1), comparing to the *optimum* obtained by GLPK using LP relaxation and integer optimization on a small scale.

Fig. 10a illustrates their comparison with the parameters of batch processing on the CPU of Galaxy S5. We can see split-shift achieves the optimum under this setting. In Fig. 10b, we use the parameters of Tegra K1 GPU on performing AlexNet, where α and β is close to γ (the time spent to extract a frame from a video). Under this setting, although split-shift continuously deviates from the optimum with the increase of the frame number, split-shift is still close to the optimum. The different performance of split-shift under these two settings can be explained intuitively as: (*i*) when detecting a frame takes much longer time than extracting a frame from a video, it is relatively easy to determine the optimal batch processing; (*ii*) when they are close, it is much more difficult to do so. This is also evidenced by GLPK, which spent much more time on finding the optimal solution under the setting of Fig. 10b than Fig. 10a for the same number of frames (minutes vs. hours). In summary, split-shift, a suboptimal algorithm with the complexity $O(n)$, is practical and affordable to solve the ILP problem, which commonly exists in optimizing the performance of the computing of deep learning.

VIII. Conclusions

In this paper, we present CrowdVision, a computing platform for crowdprocessing videos using deep learning. CrowdVision is designed to optimize the performance on mobile devices with computational offload and by taking into consideration the characteristics of the computing of deep learning for video processing. CrowdVision is implemented and evaluated on the off-the-shelf smartphone. Experimental results demonstrate that CrowdVision greatly outperforms the existing computational offload system or basic processing options under various settings and network conditions. We envision CrowdVision to be a great computing framework for crowdprocessing applications using deep learning.

References

[1] Caffe. http://caffe.berkeleyvision.org/.

[2] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *MobiSys'10*.

[3] A. Fu, P. Sadeghi, and M. Médard. Dynamic rate adaptation for improved throughput and delay in wireless network coded broadcast. *IEEE/ACM Transactions on Networking*, 22(6):1715–1728, 2014.

[4] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao. Energy-efficient computation offloading in cellular networks. In *ICNP'15*.

[5] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *MobiCom'16*.

[6] 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 *MobiSys'16*.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR'16*.

[8] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *MobiSys'12*.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS'12*.

[10] B. Liu, Y. Jiang, F. Sha, and R. Govindan. Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *SenSys'12*.

[11] Z. Lu, S. Rallapalli, K. Chan, and T. La Porta. Modeling the resource requirements of convolutional neural networks on mobile devices. In *MM'17*.

[12] M.-R. Ra, B. Liu, T. L. Porta, and R. Govindan. Medusa: A programming framework for crowd-sensing applications. In *MobiSys'12*.

[13] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *MobiSys'13*.

[14] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR'16*.

[15] D. Yang, G. Xue, X. Fang, and J. Tang. Crowdsourcing to smartphones: incentive mechanism design for mobile phone sensing. In *MobiCom'12*.