deCaffe: Crowdprocessing Videos with Cloud Offload

***Abstract*—Mobile devices such as smartphones and tablets, 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, we consider crowdprocessing videos, where each mobile device locally processes stored videos. While the computational capability of mobile devices continues to improve, video processing is still a demanding task for mobile devices. We present deCaffe, a system that enables mobile devices to crowdprocess videos using the popular deep learning platform Caffe in a distributed and efficient manner leveraging cloud offload. By carefully characterizing processing time and energy cost of video processing and offloading, deCaffe is able to quickly and efficiently process videos with offload under various settings and different network connections. We propose a split-shift algorithm to optimize completion time in WiFi networks. Under cellular, we design an adaptive algorithm that makes offloading decisions at runtime, considering energy, completion time and data budget. We implemented deCaffe on Android with a GPU-enabled server for offload and show experimentally that deCaffe significantly improves speed and energy usage of video crowdprocessing.**

I. Introduction

The widespread adoption of mobile devices and cloud has allowed for broad sensing, effortless communication, and convenient computing of information, while deep learning has enabled computers to understand information in complex formats, like image and video. These technological advances motivate us to design a cloud-based video processing system to assist in a crowdsensing application.

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 video. From a technical perspective, 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 is still a demanding task for mobile devices. 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 platform in the cloud. Coupled with the computation limitation is the cost to transmit, whether it be on WiFi or cellular networks (LTE). Transmitting over wireless interfaces consumes considerable energy. Additionally, when using cellular networks, there are finite data budgets that limit data transfer without incurring extra expenses. As a result, we consider the coupled problem of constrained resources of both battery and data transmission.

Due to these limitations, users may be reluctant to participate in these crowdsensing tasks. However, users could be motivated by various incentive mechanisms, such as [20], which are not yet our focus. 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 allow the individuals to process some of the data rather than having some centralized entity process everything.

The main idea of this paper is to consider the tradeoffs between the collective crowdprocessing performance and resource expenditures of individual user. We present *deCaffe*, a distributed, energy and data efficient video crowdprocessing system based on the popular deep learning framework Caffe [1], [11]. There are several crowdsensing frameworks [14], [13], [12] similar to our work. These frameworks provide crowdsensing capabilities to networked mobile devices. However, deCaffe is different from these frameworks in several major aspects. First, deCaffe is a video processing system rather than a system or platform that recruits users and collects sensed data; i.e., deCaffe processes existing videos on a mobile device in response to tasks. Second, deCaffe is designed to optimize task performance by considering the energy and data usage of the participating mobile devices through computation offload. The existing frameworks do not jointly (if at all) consider these factors. Moreover, for computation offload, different from the generic models, such as MAUI [5], deCaffe carefully considers the characteristic of deep learning, i.e., batch processing, which commonly exists in the computing of deep learning models.

We envision deCaffe to be useful for a variety of applications. One common case is emergency response situations, where municipal agencies ask the public to assist in identifying terrorists or criminals through scanning of their videos that may capture suspects, as the FBI did after the Boston Marathon bombing. But deCaffe could handle this in a smarter and more automatic way; i.e., it first filters videos based on location and timestamp, and then collectively performs deep learning algorithms to find the object of interest.

The main contribution of this paper is the design, implementation and evaluation of deCaffe. This contribution breaks down into the following aspects:

- We design deCaffe to enable mobile devices to crowdprocess videos using Caffe with cloud offload, while considering both the performance and resource usage.

- We design a split-shift algorithm that parallelizes frame offloading and local detection to minimize the completion time for video processing when using WiFi.
- We design an energy and data efficient algorithm that makes adaptive offloading decisions at runtime to reduce completion time and save energy for video processing using cellular networks by considering channel conditions.
- We implement and evaluate deCaffe on an off-the-shelf smartphone to confirm the performance gains for video crowdprocessing.

For the rest of the paper, we present an overview of deCaffe in Section II. Sections III and IV describe the performance optimization and proposed algorithms. We then describe the implementation in Section V and evaluation in Section VI. Related work is reviewed in Section VII, and we conclude the paper in Section VIII.

II. Overview

deCaffe is a distributed system that enables mobile devices to participate in the crowdprocessing of videos. In this environment, a task issuer initiates a query by which mobile devices are requested to identify some object of interest within its locally stored videos. We consider a crowdprocessing approach to perform object detection/classification of videos. This task includes the filtering of videos based on metadata, and then processing the videos to perform object detection. This takes several steps as described below. Caffe, 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. For this paper, we assume that the cloud and mobile devices use the same Caffe model.

Once these videos are processed (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.

Two important aspects of this process are not in the scope of this paper. First, incentive mechanisms for crowdprocessing are important but not the focus of this paper. Second, participation in the task may reveal personal information and intrude on users' privacy, but deCaffe allows users to filter out their personal videos. More sophisticated and rigorous treatment of security and privacy control is left as future work.

The overview of deCaffe 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 smartphone), or in the cloud, or distributed between the two systems.

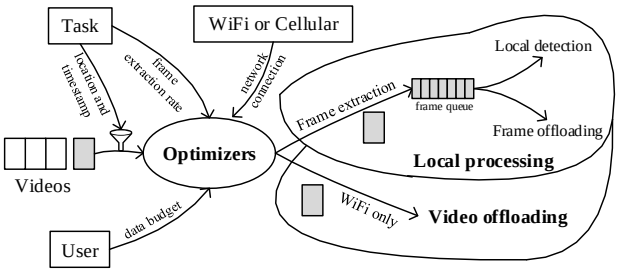


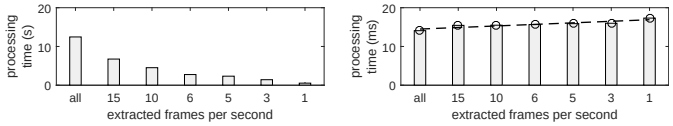
Fig. 1: Overview of deCaffe.

As illustrated in Fig. 1, by considering several input parameters and constraints, a mobile device can perform 1) *video offloading* by offloading the whole video to the cloud. Or, the user may consider performing frame extraction locally and then offloading specific frames to the cloud. In this case, it needs to determine whether each frame is processed by 2) *frame offloading* in which the frame is sent to the cloud for detection or 3) *local detection* where the frame is detected on the mobile device.

In deCaffe we consider both the task issuer's query requirements and the resource usages of mobile users. From the perspective of the task issuer, the quality of deCaffe'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 contains the event, which is determined by the deep learning model. In this work, we consider the timeliness. From the users' perspective, its primary criteria is the resource efficiency. Moreover, although the computational capability of mobile devices continues to improve, the video processing performance on mobile devices is still limited and resource intensive. Therefore, the cloud is provided by task issuers to assist in video processing to offload computational demands.

When mobile devices perform video processing with offloading, 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 are usually at home or in office where mobile devices can be easily recharged), users may not care about the resource usage as long as video processing does not affect the normal use of their mobile devices. But when using cellular networks, battery life and data usage may become the primary concerns since it may not be convenient to recharge mobile devices and cellular data plans are limited. Therefore, the design of deCaffe takes both of these into consideration.

Through the characterization of processing time, energy, and cellular data usage for video processing, we design deCaffe. 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



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

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

parameters. deCaffe needs to determine 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 remainder of this section, we describe each of these processing components and show that performing deep learning of videos is feasible on mobile devices.

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. deCaffe uses frame extraction rate as a parameter defined by the task, denoted as e_r frame per second.

Figures 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 a Galaxy S5). As shown in Fig. 2a, the total processing time for extracting all the frames takes about 13 seconds, and it decreases with the extraction rate decreases. The processing time per frame, as shown in Fig. 2b, slightly and linearly increases as the extraction rate decreases.

Moreover, as shown in Table I, the power of frame extraction using DSP is slightly more than 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 use a deep learning model (i.e., convolutional neural networks) on Caffe to perform classification of 1000 objects. Although Caffe can be accelerated by CUDA-enabled GPUs [2], it is not supported by GPUs of smartphones. Therefore, we consider offloading video/frames to the cloud for detection. 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.

The processing time of detection is affected by the size of the processing batch, 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. That means if we process ten frames one by one, it takes about 6.4s, while if ten frames are processed in a batch, it takes about 4.24s. This difference grows with the increase of the number of frames. Therefore, for detection, it is better to process frames in a batch to reduce processing time. The characteristic of batch processing commonly exists in the computing of deep learning models, e.g., convolutional neural networks, and it drives the design of deCaffe's offloading module since none of existing offloading models can directly apply to this.

	Power	**CPU usage**
Frame Extraction (DSP)	1178mW	4%
Object detection (CPU)	2191mW	25%

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

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%. 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 offloading and local processing. When choosing local processing, they then decide between frame offloading and local detection. These decisions are all affected by network conditions that lead to divergent throughput and different power. More importantly, mobile devices can be connected to the cloud via either WiFi or cellular. Each has different characteristics, and users may also have different concerns in each scenario. Therefore, we separately consider these two scenarios throughout the design of deCaffe.

III. Processing under WiFi

When mobile devices are connected to the cloud with WiFi, as discussed above, resource usage may not be a concern for users. Therefore, we focus on the optimization of the processing time on a mobile device.

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. As videos are processed in serial, optimizing the completion time of all

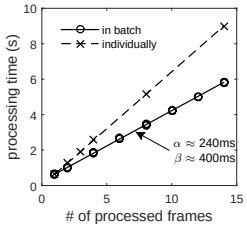


Fig. 3: Processing time of detection performed individually or in batch on smartphone.

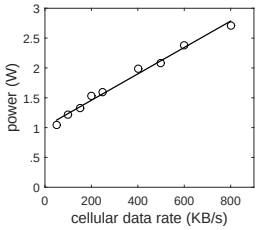


Fig. 4: Power of LTE in terms of different uplink rates on smartphone.

related videos on a mobile device is equal to minimizing the completion time of each video. Therefore, for each video, we need to estimate and compare the completion time for video offloading and local processing.

Mobile devices are usually connected to WiFi when users are at home or at work. In such environments, the channel conditions and network traffic varies only slightly. Therefore, for the processing under WiFi, the data rate between a mobile device and cloud can be considered to be stable. Let r denote the data rate. Note that the environment with a dynamic WiFi data rate can be handled by the adaptive algorithm discussed in Section IV without imposing a data usage constraint.

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

Processing Time on Frame Extraction. As shown in Figure 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) may 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.

Processing Time on Local Detection. As discussed in Section II-B and 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. We discuss a formulation below.

Assume there will be 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 determine how to process these frames, i.e., how many processing batches are needed and how many frames each batch should process. First, let us assume the optimal number of batches is m. Then, we need to decide how many frames should be processed for each batch.

Let y_i denote the number of frames for batch $i \in [1, m]$, x_i denote the time interval between the start times of batch i and $i+1$, and x_0 denote the waiting time before processing the first batch. Before processing each batch i, the frames of batch i should already be available and the previous batch $i-1$ must already have been completed. Thus, the minimal processing time of detection on n extracted frames by m batches can be formulated as

$$\begin{aligned} \min \quad & \sum_{i=0}^{m} x_i \\ \textit{s.t.} \quad & x_k \geq \alpha + \beta y_k, \qquad k = 1, \ldots, m \\ & \sum_{i=0}^{k} x_i \geq \sum_{i=1}^{k+1} \gamma y_i, \quad k = 0, \ldots, m-1 \\ & y_k > 0, \qquad\qquad k = 1, \ldots, m \\ & \sum_{i=1}^{m} y_i = n. \end{aligned} \tag{1}$$

To obtain the minimal processing time of m batches on n frames, we have to solve the Integer Linear Programming problem (1). However, since $m = 1, \ldots, n$, the optimal solution of minimizing the processing time on detection costs too much, even for a small n.

Processing time on Frame Offloading. Besides detecting extracted 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 $\delta = \frac{d_f}{r}$ and d_f is the data size of an extracted frame. Note that the extracted frame may be resized to feed different object detection algorithms 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 $\delta > \gamma$, there will be frame backlog and local detection is needed. However, as discussed above, minimizing the processing time of frame 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 in the following to solve the problem.

B. The Split-Shift Algorithm

For each extracted frame, we have two options: frame offloading or local detection, which are two processes working in parallel to reduce the processing time for 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, 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.

For offloading, it is better to keep busy sending frames. If only frame offloading 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 offloading 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 affordable for mobile devices. Moreover, it is also easy to be implemented on mobile devices; i.e., the offloading process simply keeps offloading 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 offloading and local processing, respectively, and then chooses the approach that has a better completion time.

IV. 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 is not 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 tend to 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 offloading 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 trade-offs between these two objectives. However, due to the variation of cellular data rate, we cannot solve the problem traditionally by Pareto optimal solutions. Thus, we have to design an adaptive algorithm that makes a decision on each extracted frame (i.e., between frame offloading and local detection) towards optimizing both objectives at runtime.

To perform video processing under cellular, users must specify a data usage constraint D' between zero and a maximum, which can be easily calculated based on video specifications, frame extraction rate and frame data size. When D' is equal to zero, the problem is trivial and all extracted frames are detected locally. In the following, we consider the case that D' is greater than zero.

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 offloading. 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 consumption of an LTE uplink exhibits an approximately linear relationship with data rates as depicted

in Fig. 4, similar to the result in [10]. 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 offloading. 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. The 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 offloading and local detection. Frame offloading may improve completion time or energy expenditure 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.

For these two options, frame offloading 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 offloading. In addition, we do not parallelize frame offloading and local detection, since this always sacrifices one objective for another. Based these considerations, we design the 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 had local detection been performed, this offloading is called a successful attempt; otherwise, it is referred to as an unsuccessful attempt.

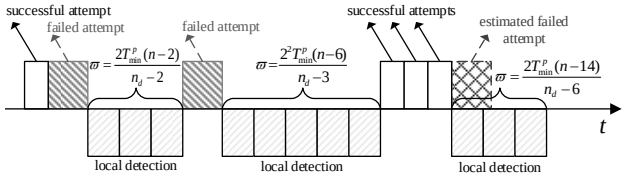


Fig. 5: Illustration of video processing under cellular.

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 offloading (i.e., frame offloading 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), that can be completed within ω. After a timeout of the backoff timer, the mobile device will switch back to frame offloading. The process iterates until frame offloading 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 offloading and local detection are performed alternately to find the times when frame offloading 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.

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 $\omega = \frac{2T_{\min}p(n-2)}{n_d-2}$ 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

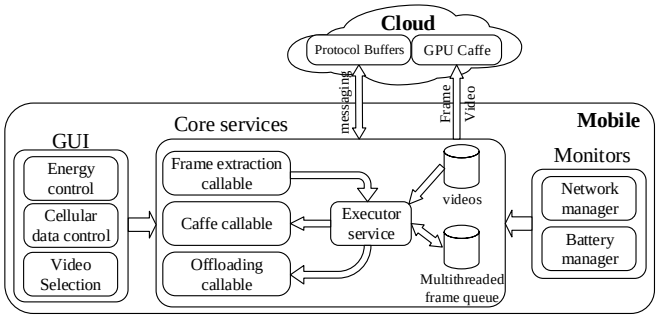


Fig. 6: Architecture of deCaffe.

is set to $\frac{2^2 T_{\min} p(n-6)}{n_d - 3}$ since there are two consecutive attempt failures. After switching back to frame offloading, 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 offloading and local detection, we choose to perform them alternatively. Frame offloading is employed to optimize both completion time and energy consumption. However, due to the variation of network conditions, frame offloading 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.

V. Implementation

We have implemented deCaffe 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 deCaffe consists of three components: core services, monitors and a GUI, as illustrated in Fig. 6.

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 deCaffe can quickly screen videos for the processing task. The Multithreaded frame queue stores extracted frames from videos and supports multithreading access. The executor servce is able to call any of the callables to perform frame extraction, Caffe for 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 the core services. The network monitor tells the cores services current network connection with distinct metrics for WiFi or cellular networks. For WiFi, it measures the data rate between a mobile device and cloud using a sample file. 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 cellular data usage and access to videos. For cellular, the data usage limits can be specified by users. Additionally, users can select videos to not be processed by deCaffe, for privacy concerns or any other manual filtering requirements.

VI. Evaluation

In this section, we investigate the performance of the proposed algorithms and show experimental results for the system implementation.

A. Algorithm Performance

We evaluate the algorithms by using empirically gathered measurements of processing time and energy taken from a Galaxy S5 (from Section II). We use these performance metrics in simulations to investigate the impact of input parameters, such as WiFi/cellular data rate, frame data size and frame extraction rate, for the different processing options. The simulation is carried out on a 1080p 30-second video. Note that the video specification and duration do not affect the relative performance of different processing options.

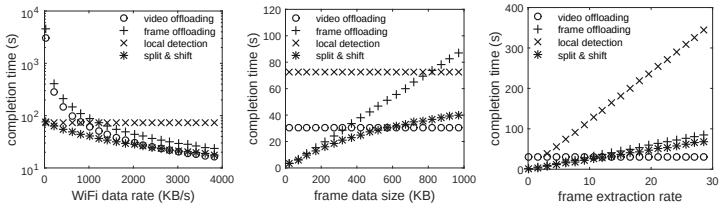
WiFi. Fig. 7a illustrates the completion time when solely using video offloading, solely frame offloading, solely local detection and *split-shift* (i.e., the completion time produced by the split-shift algorithm) in terms of WiFi data rate, where the frame data size d_f is 500kB and frame extraction rate r_e is 6fps. When the WiFi data rate is low, split-shift performs much better than video offloading. The WiFi data rate does not affect local detection and thus its performance is constant. When the WiFi data rate is high (i.e., more than 3MB/s), video offloading is slightly better than *split-shift*. Since *split-shift* parallelizes frame offloading and local detection to improve completion time, it performs better than other two individually.

Since Caffe models may require different resolutions of images as input, we also investigate the effect of frame data size on the completion time. As shown in Fig. 7b, with the increase of frame data size, more data must be sent to the cloud for frame offloading and *split-shift*, and thus their completion times both increase. Meanwhile, for video offloading and local detection, the completion time is unchanged as depicted in Fig. 7b.

To detect different objects, different frame extraction rates may be defined by the task issuer. Fig. 7c depicts the effect of frame extraction rates on the completion time. An increased frame extraction rate increases completion time for frame offloading, local detection and *split-shift*, since more frames are to be locally processed. Note that due to video compression, if every single frame is extracted, the size of all of the extracted frames could be much larger than the video itself. Thus, when frame extraction rate and WiFi data rate are both high, it is better to perform video offloading.

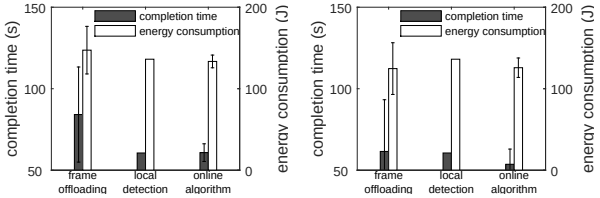
For video processing under WiFi, we can estimate the completion time for each processing option and make the selection for the minimal completion time.

Cellular. Based on the measurements of the LTE module on a Galaxy S5 under different uplink rates, as illustrated



(a) (b) (c)

Fig. 7: Completion time in terms of WiFi data rate (a), frame data size (b), and frame extraction rate (c), where the default values are $d_f = 500$KB/s, $r_e = 6$fps, and $r = 2000$KB/s.



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

Fig. 8: Performance under different cellular data rates, where $d_f = 100$KB, $r_e = 6$fps.

in Fig. 4, we perform the evaluation for the processing under cellular. We compare the adaptive algorithm with frame offloading and local detection in terms of completion time and energy.

To model the dynamics of cellular data rate, we also adopt a Markov chain [6]. 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

$$P = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccccc} r_0 & r_1 & r_2 & r_3 & r_4 \end{array} \\ \begin{bmatrix} 2/3 & 1/3 & 0 & 0 & 0 \\ 2/9 & 5/9 & 2/9 & 0 & 0 \\ 0 & 2/9 & 5/9 & 2/9 & 0 \\ 0 & 0 & 2/9 & 5/9 & 2/9 \\ 0 & 0 & 0 & 1/3 & 2/3 \end{bmatrix} \end{array}.$$

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 the adaptive algorithm to half of total extracted frames.

Figures 8a and 8b give the performance when using solely frame offloading, solely local detection and the adaptive algorithm under R_1 and R_2, respectively. As shown in Fig. 8a, the adaptive algorithm outperforms frame offloading in terms of both completion time and energy; its performance is similar to local detection. When cellular data rates increase to R_2, as shown in Fig. 8b, frame offloading performs better than before as expected. Since cellular data rates do not affect local detection, its performance remains the same. The adaptive algorithm performs the best. It has much less completion time than others and the same energy consumption as frame offloading.

The adaptive algorithm outperforms frame offloading and local detection under different cellular data rates, because it 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 of the adaptive algorithm avoids unnecessary frame offloading when the cellular data rate is low.

B. System Performance

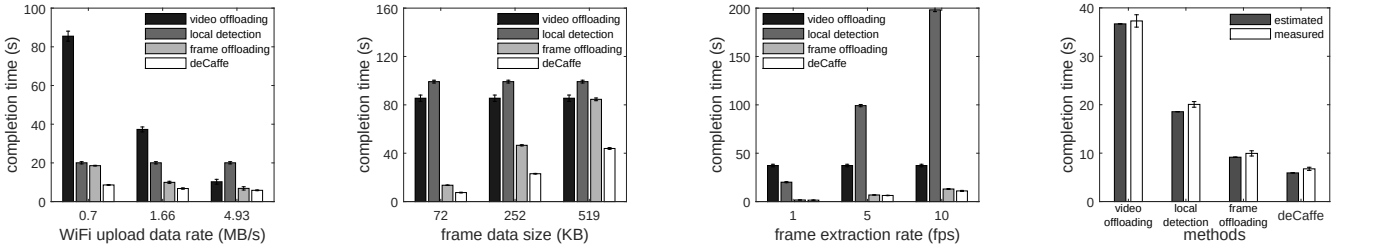
Using our implementation of deCaffe, we conducted experiments to evaluate the performance of deCaffe on a Galaxy S5 in terms of completion time and energy, where energy consumption is measured by a power monitor. We conduct experiments using WiFi and cellular networks.

WiFi. Fig. 9 illustrates the completion time of different methods for a 1080p 30-second video under various setting and WiFi data rates. The experiments are conducted using the following parameters: upload data rates (0.7, 1.66 and 4.93)MB/s, frame data sizes (72, 252 and 519kB, which are the sizes of JPG with resolutions 640×360, 1280×720 and 1920×1080, respectively) and frame extraction rates (1, 5, 10)fps. Figures 9a, 9b and 9c depict the completion time in terms of WiFi upload data rate, frame data size and frame extraction, respectively. The results are similar to Fig. 7 and further verify the correctness of the estimation of completion time in terms of these parameters. Moreover, the completion time for each method should be accurately estimated in order to choose the approach for the best performance. As shown in Fig. 9d, the estimated completion time is very close to the measured completion time. Therefore, deCaffe can reliably make the best decision based on this estimate.

Cellular. Experiments for cellular were carried out at three different locations to acquire different uplink data rates. As illustrated in Fig. 10, when the data rate is low, deCaffe outperforms frame offloading in terms of both completion time and energy. Since deCaffe 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, deCaffe and frame offloading perform better than before. deCaffe has similar completion time with local detection but uses less energy. When the data rate further increases, deCaffe is faster and uses less energy than the others. When the data rate is sufficiently high such that offloading a frame always outperforms locally detecting a frame in both completion time and energy cost, deCaffe will also offload all the frames to the cloud as frame offloading does. Therefore, deCaffe adapts to different cellular data rates to reduce completion time and save energy.

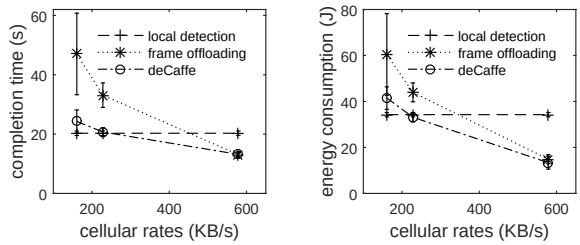
VII. Related Work

There are several examples of crowdsensing frameworks similar to deCaffe. Medusa [14] is a platform that performs crowdsensing tasks through the transfer and processing of



(a) $d_f = 519$KB, $r_e = 1$fps (b) $r = 0.7$MB/s, $r_e = 5$fps (c) $r = 0.7$MB/s, $d_f = 252$KB (d) $r = 1.66$MB/s, $d_f = 519$KB, $r_e = 1$fps

Fig. 9: System performance of different processing options with various settings and different WiFi data rates.



(a) completion time (b) energy consumption

Fig. 10: System performance of different processing options under different cellular uplink rates, where $d_f = 252$KB, $r_e = 1$fps.

media (text, images, videos). Pickle [13] is a crowdsensing application to conduct collaborative learning. GigaSight [19] is a cloud architecture for continuous collection of crowd-sourced video from mobile devices. In contrast to these frameworks, deCaffe is a video processing system rather than the system or platform that motivates users and collects sensed data. For our crowdprocessing task, deCaffe attempts to optimize the performance of the task execution and also considers the energy usage and data usage of the participating mobile devices through computation offload.

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 [5], [4], [18] and computation offload based applications such as [16], [15], [9], [7]. However, due to the characteristic (batch processing) of deep learning based video processing, none of these works can be directly applied to our application and offloading problem.

A potential option to augment the performance of computer vision algorithms for mobile devices is on-board GPUs. Recent work [8] considers the difficulty of running deep neural networks (DNNs) on mobile devices, while the feasibility of running DNNs on GPUs of Tegra TK1 boards is investigated in [17]. TensorFlow [3] is a recently released library that provides the ability to setup distributed computation across networks of devices and over CPUs and GPUs. However, currently, these approaches require custom hardware or software. Neverthless, these potential processing approaches can be easily integrated with deCaffe for video crowdprocessing whenever they are available for off-the-shelf mobile devices.

VIII. Conclusions

In this paper, we present deCaffe, a video crowdprocessing system with computational offload that optimizes performance based on energy usage and data usage. deCaffe is able to identify the offload strategy with regard to channel conditions and energy usage on mobile devices. deCaffe is implemented and evaluated on an off-the-shelf smartphone. Experimental results demonstrate that deCaffe can greatly perform video processing with cloud offload under various settings and network conditions.

References

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

[2] http://www.nvidia.com/.

[3] http://www.tensorflow.org/.

[4] M. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *INFOCOM*, 2013.

[5] 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*, 2010.

[6] 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.

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

[8] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An execution framework for deep neural networks on resource-constrained devices. In *MobiSys*, 2016.

[9] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. Weardrive: fast and energy-efficient storage for wearables. In *ATC*, 2015.

[10] 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*, 2012.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *MM*, 2014.

[12] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan. MediaScope: Selective On-Demand Media Retrieval from Mobile Devices. In *IPSN*, 2013.

[13] B. Liu, Y. Jiang, F. Sha, and R. Govindan. Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *SenSys*, 2012.

[14] M.-R. Ra, B. Liu, T. L. Porta, and R. Govindan. Medusa: A programming framework for crowd-sensing applications. In *MobiSys*, 2012.

[15] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *MobiSys*, 2012.

[16] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. Sociablesense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *MobiCom*, 2011.

[17] S. Rallapalli, H. Qiu, A. J. Bency, S. Karthikeyan, R. Govindan, B. S. Manjunath, and R. Urgaonkar. Are very deep neural networks feasible on mobile devices? In *HotMobile*, 2016.

[18] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura. Cosmos: computation offloading as a service for mobile devices. In *MobiHoc*, 2014.

[19] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *MobiSys*, 2013.

[20] D. Yang, G. Xue, X. Fang, and J. Tang. Crowdsourcing to smartphones: incentive mechanism design for mobile phone sensing. In *MobiCom*, 2012.