DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics

Xukan Ran*, Haoliang Chen*, Xiaodan Zhu†, Zhenming Liu†, Jiasi Chen*
*University of California, Riverside, Riverside, CA †College of William and Mary, Williamsburg, VA
{xukan.ran, haoliang.chen}@email.ucr.edu; xzhu08@email.wm.edu; zliu@cs.wm.edu; jiasi@cs.ucr.edu

***Abstract*—Deep learning shows great promise in providing more intelligence to augmented reality (AR) devices, but few AR apps use deep learning due to lack of infrastructure support. Deep learning algorithms are computationally intensive, and front-end devices cannot deliver sufficient compute power for real-time processing. In this work, we design a framework that ties together front-end devices with more powerful backend "helpers" (*e.g.,* home servers) to allow deep learning to be executed locally or remotely in the cloud/edge. We consider the complex interaction between model accuracy, video quality, battery constraints, network data usage, and network conditions to determine an optimal offloading strategy. Our contributions are: (1) extensive measurements to understand the tradeoffs between video quality, network conditions, battery consumption, processing delay, and model accuracy; (2) a measurement-driven mathematical framework that efficiently solves the resulting combinatorial optimization problem; (3) an Android application that performs real-time object detection for AR applications, with experimental results that demonstrate the superiority of our approach.**

I. Introduction

Deep learning shows great promise to provide more intelligent video analytics to augmented reality (AR) devices. For example, real-time object recognition tools could help users in shopping malls [1], facilitate rendering of animations in AR apps (*e.g.,* detect a table, and overlay a game of Minecraft on top of it), assist visually impaired people with navigation [2], or perform facial recognition for authentication [3].

Today, however, only a few AR apps use deep learning due to insufficient infrastructure support. Object recognition algorithms such as deep learning are the bottleneck for AR [4] since they are computationally intensive, and the front-end devices are often ill-equipped to execute them with acceptable latencies for the end user. For example, Tensorflow's Inception deep learning model can process about one video frame per second on a typical Android phone, preventing real-time analysis [5]. Even with speedup from the mobile GPU [6], [7], typical processing times are approximately 600 ms, which is less than 1.7 frames per second. In industry, while a few applications run deep learning locally on a phone (*e.g.,* Apple Photo Album), these are lightweight models that do not run in real time. Voice-based intelligent personal assistants (*e.g.,* Alexa, Cortana, Google Assistant, and Siri) mostly transfer the input data to more powerful backends and execute the deep learning algorithms there. Such cloud-based solutions are only applicable when network access is reliable.

Our observations. While front-end devices are computationally weak, and sending deep learning jobs to "backend" computers is inevitable, the following new observations will yield an effective design.

Tradeoffs between accuracy and latency. AR apps relying on deep learning have different accuracy/latency requirements. For example, AR used in shopping malls for recommending products may tolerate longer latencies (fine to let users to wait a second or two) but have a higher accuracy requirement. In an authentication system that uses deep learning, users could wait even longer but expect ultra-high accuracy.

Sources of latency. When a deep learning task needs to be executed remotely, both the data transmission time over the network and the deep learning computation time can introduce latencies. Prior works (*e.g.,* [7], [8]) focus on optimizing *computation latencies* (*i.e.,* time between the job's arrival at the computation node and the job's completion) by designing sophisticated local scheduling algorithms. We observe that network latencies are often much longer than the computation latencies, so it is important to optimize the offloading decision along with the local processing. Furthermore, although deep learning based real-time video analytics are known to be computationally intense, simple consumer-grade GPUs suffice for most real-time video analysis (*e.g.,* object detections). Thus, home computers could be used as "backend helpers" that are dedicated to a small group of users like family members. In other scenarios, home desktops may not even be needed, as wearable devices such as smartwatches or head-mounted displays could send computation to a user's smartphone nearby.

Video streams and deep learning models as "first class citizens." Deep learning in an AR setting is primarily responsible for interpreting data collected from a camera (*i.e.,* video data). Prior works (*e.g.,* [5], [7], [8]) treat the videos merely as sequences of images and the deep learning models as rigid computation devices that produce uniform forecasting quality. Yet these assumptions lead to the illusion that we face a canonical scheduling problem: a fixed set of computation tasks needs solving, and each deep learning model consumes a predictable amount of resources and produces predictable output (*i.e.,* forecasting quality is known). The assumption, however, will substantially reduce the system performance because they ignore the compressibility of both deep learning models and video streams.

Instead, we ought to treat video streams and deep learning modules as "first-class citizens" and directly optimize the tradeoffs between video and prediction qualities. Specifically, video data should not be treated as a sequence of images (*i.e.,* independent computation tasks) because this will over-consume network bandwidth; instead, aggressive leverage of existing technologies for compressing videos (including DFT, delta coding, and changing resolution *etc.*) will result in the best use of network bandwidth. Certainly, over-aggressive compression may cause declines in video analysis quality. Our solution aims to find the most suitable video encoding scheme that gives the optimal tradeoff between network consumption and prediction quality.

Our contribution. We propose a distributed infrastructure, DeepDecision, that ties together computationally weak front-end devices (assumed to be smartphones in this work) with more powerful back-end helpers to allow deep learning to choose local or remote execution. The back-end helpers can be any devices that supply the requisite computation power. Our solution intelligently uses current estimates of network conditions, in conjunction with the application's requirements and specific tradeoffs of deep learning models, to determine an optimal offload strategy. In particular, we focus on executing a convolutional neural network (CNN) designed for detecting objects in real-time for AR applications. (A similar framework could be applied to any application that requires real-time video analytics.) We seek to understand how the changes of key resources (*e.g.,* network bandwidth, neural network model size, video resolutions, battery usage) in the system impact the decision of where to compute. An overview of our system is illustrated in Fig. 1.

We make the following contributions:

1. Extensive measurements of deep learning models on smartphones to understand the tradeoffs between video compression, network conditions and data usage, battery consumption, processing delay, frame rate, and machine learning accuracy;

2. A measurement-driven mathematical framework that efficiently solves an optimization problem, based on our understanding of how the above parameters influence each other;

3. An Android implementation that performs real-time object detection for AR applications, with experiments that confirm the superiority of our approach compared to several baselines.

Organization. Sec. II explains the background on neural networks, Sec. III describes our model and algorithm, and Sec. IV shows our measurements and experimental results. Finally, Sec. V discusses related work and Sec. VI concludes.

II. Background, Metrics, and Degrees of Freedom

Background. We first provide relevant background on CNNs for video analytics and AR. In video analytics, object recognition (classifying the object) and object detection (locating the object in the frame) are both needed. In AR, the processing pipeline also includes drawing an overlay on top of the located and classified object. Neural nets are the state-of-the-art in computer vision for object recognition and detection,

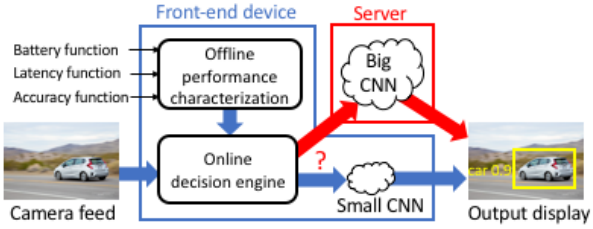


Fig. 1: System overview. The front-end device chooses where to analyze the input video for real-time display.

and many existing neural nets for object recognition build a pipelined solution, *i.e.,* they use one neural net to detect the boundaries of objects and a second net to inspect contents within each bounding box. In this work, we use a particular CNN called Yolo [9]. (Our framework can also be adapted to other popular CNNS such as [10], [11].) Yolo is optimized for processing video streams in real-time and possesses two salient features: 1. *One neural net for boundary detection and object recognition.* Observing that using multiple neural nets unnecessarily consumes more resources, Yolo trains one single neural network that predicts boundaries and recognizes objects simultaneously. 2. *Scaling with resolution.* Yolo handles images with different resolutions, *e.g.,* when there is a change in the dimension of an input to a convolutional layer, Yolo does not change the kernels and their associated learnable parameters – this would result in a change of output dimensions. Thus, the compute time of Yolo scales directly with input's resolution, *e.g.,* lower resolution images require less computation.

Key performance metrics. AR apps often require service guarantees on two important metrics:

1. Frame rate: The frame rate is the number of frames that we feed the deep learning model per second when it's running locally, or is the video frame rate (FPS) when offloading.

2. Accuracy: The accuracy is a metric that captures (a) whether the object is classified correctly; (b) whether the location of the object in the frame is correct.

Being a responsible citizen. While purely focusing on these key metrics may maximize the performance of the video analytics module, other potential impacts on the frontend device should be considered. For example, running more powerful deep learning models will consume extra CPU cycles, disrupting other background processes, and draining the device battery. If the client communicates with the server, the network transmission also uses battery; also, if the data transfer is over LTE, the monetary cost to the user in terms of data quota must also be considered. These factors of battery consumption and network data usage must be considered holistically alongside the key performance metrics.

Degrees of freedom. There are several degrees of freedom we consider in this work.

1. Adjust the frame resolution. By decreasing the frame resolution, we can decrease the execution time of a deep learning model, which also lowers the energy cost. However,

Variable	Description
p	frame resolution (pixels^2)
r	video bitrate (bits/s)
f	frame rate (frames/s)
$y_i(t)$	model decision at time t
$a_i(p, r, \ell_i)$	accuracy of model i (%)
$b_i(p, r, f)$	battery of model i (J/s if $i = 0$, J/frame if $i > 0$)
$\ell_i(p, r, f)$	total delay when using model i (s/frame)
$\ell_i^{\text{CNN}}(p)$	processing delay from running model i
B	network bandwidth (kbps)
L	network latency (s)
$\mathcal{B}$	target battery usage (J/s)
A	accuracy target (%)
F	frame rate target (frames/s)
c	monetary cost ($/bit, if use cellular network)
C	target monetary cost ($/bit)
α	parameter that trades off accuracy for frame rate in the objective function

TABLE I: Table of Notation. $i = 0$ represents remote execution, and $i = 1, \ldots, N$ represents local execution on the front-end device.

this may also decrease the accuracy of the model. Conversely, increasing the frame resolution may increase the accuracy, at the expense of lower frame rate and greater energy drain.

2. Use smaller deep learning models. We may wish to use a smaller neural network to reduce the run time and the energy cost, at the cost of reducing the prediction accuracies. Conversely, using a larger deep learning model increases the run time and energy, but boosts the accuracy.

3. Offload to backend. By sending the computation job to a backend server, we can substantially reduce the computational burden at front-end devices, increasing the frame rate. However, this may result in extra startup delay from the network transfer, causing the server's result to be stale by the time it is returned to the client, thus decreasing the prediction accuracy.

4. Compress the video. When offloading, one may wish to compress the video more/less based on the network conditions. Choosing a low target video bitrate reduces the network transmission time, but potentially decreases the accuracy. Conversely, using a high video bitrate may result in higher accuracy but will also result in longer transmission time, making the detection results stale and decreasing the accuracy.

5. Sample the video at lower frequencies. We do not need to process every frame in the video; instead we may sample only a small fraction of frames for further processing. In this way, we may reduce the total computation demand of an AR app.

Each of these operations may impact one or more of the key performance metrics described above. Furthermore, we may employ multiple operations simultaneously, *e.g.,* we can reduce the resolution and use smaller models at the same time. In fact, any subset of these operations defines a legitimate strategy, although not necessarily optimal.

III. Our algorithmic problem

This section describes the problem, our optimization framework, and the algorithms to solve this optimization problem. **Challenges:** (1) As can be seen from the previous section, the interactions between the degrees of freedom and the key performance metrics are complex. For example, some decision variables increase one key metric but decrease another metric (*e.g.,* higher resolution increases accuracy but decreases frame rate); or some decision variables may affect the same metric in multiple ways (*e.g.,* transmitting the video at a higher bitrate could increase the accuracy, but could also increase the latency, which decreases accuracy. See Fig. 5). Selecting the right combination of decision variables that maximizes the key metrics, while satisfying energy, cost, and performance constraints is no easy task. (2) Moreover, many of these tradeoffs cannot be expressed cleanly in analytic form, making any solution or analysis difficult. For example, analyzing the relationship between staleness and accuracy is a challenging task that depends on the video content, the particular deep learning model being used, the resolution of the video, and the compression of the video. The lack of analytic understanding of these tradeoffs is in part due to the complexity of the deep learning models themselves, whose theoretical properties are not yet well understood.

Our approach: Our approach is therefore to create a *data-driven* optimization framework that takes as input the empirical measurements of these tradeoffs, computes the optimal combination of decision variables that maximizes the key metrics, and outputs the optimal decision. Our framework must be general enough to handle any values of input measurement data while still capturing the tradeoffs between decision variables and metrics. In this section, we will describe the optimization framework that we designed and its solution; while Sec. IV, we show the actual input data based on our measurements with real systems, as well as the experimental results from our Android application.

In the DeepDecision system, we divide time into windows of equal size, and solve an optimization problem at the beginning of each interval to decide the deep learning algorithm's configurations: the frame rate sampled by the camera (f), the frame resolution (p), the video bitrate (r), and which model variant to use (y_i). The problem is:

Problem 1:

$$\underset{p,r,f,\mathbf{y}}{\text{maximize}} \quad f + \alpha \left(\sum_{i=0}^{N} a_i(p, r, \ell_i) y_i \right) \tag{1}$$

$$\text{s.t.} \quad \ell_i = \begin{cases} \ell_i^{\text{CNN}}(p) + \frac{r}{f \cdot B} + L & \text{if } i = 0 \\ \ell_i^{\text{CNN}}(p) & \text{if } i > 0 \end{cases} \tag{2}$$

$$\sum_{i=0}^{N} \ell_i^{\text{CNN}}(p) y_i \leq 1/f \tag{3}$$

$$r \cdot y_0 \leq B \tag{4}$$

$$\sum_{i=0}^{N} b_i(p, r, f) y_i \leq \mathcal{B} \tag{5}$$

$$c \cdot r \cdot y_0 \leq C \tag{6}$$

$$f \geq F \tag{7}$$

$$\forall i : a_i(p, r, f) \geq A \cdot y_i \tag{8}$$

$$\sum_{i=0}^{N} y_i = 1 \tag{9}$$

$$\text{vars} \quad p, r, f \geq 0, y_i \in \{0, 1\} \tag{10}$$

The objective (1) is to maximize the number of frames sampled plus the accuracy of each frame (see Key Performance

Metrics in Sec. II). The relative importance of accuracy versus frame rate is determined by parameter α. Constraint (2) says that the total delay experienced by a frame is equal to the CNN's processing time plus the network transmission time (if applicable). Constraint (3) says that the frame rate cannot be chosen to exceed the processing time of the CNN (remote or local). Constraint (4) says that the video cannot be uploaded at a higher bitrate than the available bandwidth. Constraint (5) says that the battery usage cannot exceed the maximum target. Constraint (6) says that the monetary cost cannot exceed the maximum target. Constraints (7) and (8) allow the application to define a minimum required frame rate and accuracy. Constraint (9) says that only one model may be selected (remote or one of the local models).

If the frame rate, resolution, and video bitrate were known, then Prob. 1 is a multiple-choice multiple-constraint knapsack program, where the items are the model variants, an item's utility is the model's accuracy, and an item's weight is in terms of latency, bandwidth, battery, and monetary cost. The multiple-choice comes from the fact that for each frame, there is a choice to offload, or process locally (choosing a model size). The multiple-constraint comes from the latency, bandwidth, battery, and cost constraints. However, the key difference from the classical problem is that the utility and costs of the items are also functions of the optimization variables. Moreover, they are generally non-linear functions, and must be determined empirically from measurements.

A brute-force solution to Prob. 1 would take $O(r_{\max} \cdot f_{\max} \cdot p_{\max} \cdot N)$. The brute-force solution is impractical when we need to frequently make new decisions and/or carry out a grid-search in fine granularity. We need to leverage the mathematical structure of the problem (based on simple intuitions and confirmed by extensive measurements in Sec. IV) to design a more efficient algorithm. These intuitions are:

1. Accuracy: For remote models, the accuracy per frame depends on a number of factors: the video resolution, the video bitrate/compression, and the end-to-end delay (which is itself a function of resolution, bitrate, frame rate, and network latency and bandwidth), i.e. $a_0(p, r, \ell_0) = a_0(p, r, \ell_0(p, r, f, B, L))$. For local models, the accuracy model can be simplified because it depends only on the resolution and delay, since we do not need to compress the video for transmission over the network, *i.e.,* $a_i(p, r, \ell_i) = a_i(p, \ell_i), i > 0$.

2. Battery: When transmitting to the server, the energy per time depends only on how much data is transmitted over the network, *i.e.,* $b_0(p, r, f) = b_0(r, B)$. For local models, the battery usage per time depends on the resolution and the number of frames processed, *i.e.,* $b_i(p, r, f) = f \cdot b_i(p), i > 0$.

3. Latency: The latency per frame when running the CNN locally depends only on the resolution, since a larger resolution requires more convolutions to be performed, *i.e.,* $\ell_i(p, r, f) = \ell_i^{\text{CNN}}(p), i > 0$. When offloading to the server, however, the total latency is a function of the CNN processing time plus the network transmission time, *i.e.,* $\ell_0(p, r, f) = \ell_0^{\text{CNN}}(p) + \frac{r}{f \cdot B} + L$.

We are now ready to describe our Alg. 1 that exactly solves

Algorithm 1 DeepDecision algorithm

Input: Target cost C, target battery $\mathcal{B}$, cost per bit c, network bandwidth B, network latency L, model battery usage function b_i, model latency function ℓ_i, model accuracy function a_i
Output: Frame resolution p^*, video bitrate r^*, frame rate f^*, decision of model variant $\mathbf{y}^*$

1: $f \leftarrow \frac{1}{f_0^{\text{CNN}}}$ ▷ remote model
2: $p, r \leftarrow \arg\max_{p, r \le \min(B, b_0^{-1}(\mathcal{B}|B), \frac{C}{c})} (f + a_0(p, r, \ell_0^{\text{CNN}} + \frac{r}{fB} + L))$
3: **if** $a_0(p, r, f) \ge A$ **and** $f \ge F$ **then**
4: $u_{\max} \leftarrow f^* a_0(p, r, f)$
5: $r^* \leftarrow r, f^* \leftarrow f, p^* \leftarrow p, \mathbf{y}^* \leftarrow e_i$
6: **for** $i \leftarrow 1$ to N **do** ▷ try the local models
7: $r \leftarrow r_{\max}$ ▷ don't need to compress locally
8: $p \leftarrow \arg\max_p (\min(\frac{1}{\ell(p)}, \frac{\mathcal{B}}{b(p)}) + a_i(p, \ell_i(p))$
9: $f \leftarrow \min(\frac{1}{\ell(p)}, \frac{\mathcal{B}}{b(p)})$
10: $u \leftarrow f + a_i(p, \ell_i(p))$
11: **if** $u > u_{\max}$ **and** $a_i(p) \ge A$ **and** $f \ge F$ **then**
12: $u_{\max} \leftarrow u$
13: $r^* \leftarrow r, f^* \leftarrow f, p^* \leftarrow p, \mathbf{y}^* \leftarrow e_i$
14: **return** $p^*, r^*, f^*, \mathbf{y}^*$

Prob. 1 and improves on the brute-force search efficiency. With some abuse of notation, we define the inverse of a function $g : \mathbb{R}^2 \to \mathbb{R}$ as $g^{-1}(y|z) = \arg\max_x (g(x, z) : g(x, z) \le y)$.

- Line 1-5, $i = 0$ (remote model): The constraints are: $\ell_0 = \ell_0^{\text{CNN}} + \frac{r}{f \cdot B}, \ell_0^{\text{CNN}} \le \frac{1}{f}, b_0(r, B) \le \mathcal{B}, c \cdot r \le C$, and $r \le B$. Since the objective increases with frame rate (the f term grows and $\frac{r}{f \cdot B}$ shrinks, resulting in higher accuracy), we can pick the maximum frame rate that satisfies the constraints. We next search across resolution p and bitrate r to find the best combination. Lastly, we check if the frame rate and accuracy constraints are satisfied.
- Line 6-13, $i > 0$ (local models): The constraints are: $\ell_i(p) \le \frac{1}{f}$ and $f \cdot b_i(p) \le \mathcal{B}$. Since the processing is local, the video bitrate is set to the maximum. The tradeoff is between the resolution p and frame rate f (setting a higher resolution improves accuracy, but the CNN takes longer, decreasing the frame rate). For each value of p, since $b_i(p)$ and $l_i(p)$ are non-decreasing, we can find the maximum f that still satisfies the constraints, then pick the best p overall. Finally, we check if the frame rate and accuracy constraints are satisfied.[1]

The battery, latency, and accuracy functions $b_i^{-1}, \ell_i^{-1}, a_i$ can easily be pre-stored (by using hash tables) so that their lookups take constant time. We use linear interpolation on these functions if the measurement density is insufficient. The running time of our algorithm takes $O(p_{max}(r_{max} + N))$, which is a significant improvement over the brute-force solution.

[1]We remark that when the frame rate does not satisfy (7), we could not have picked a different frame rate resulting in a feasible solution. This is because the algorithm picks the maximum possible f for each value of p. A similar argument also holds for accuracy constraint A.

We next make a number of remarks. **1. Utility function.** The utility function in (1) is the sum of frame rate and accuracy. One may also consider other utility functions, *e.g.,* multiplicative $f \cdot \sum_{i=0}^{N} a_i(p, r, \ell_i) y_i$, where the intuition is that utility depends on collecting more frames each with high accuracy. Alg. 1 is still applicable when the utility function changes (so long as it is monotone in both frame rate and accuracy). We have experimented with the multiplicative objective function and found that it tends to emphasize frame rate at the expense of accuracy, and choose solutions with low accuracy but high frame rate. Therefore, we use the additive objective function (1) in the remainder of this work.

2. Budgets in each time interval. In the current formulation, the user inputs her battery and monetary constraints as an average usage over time (*e.g.,* $\$/s, J/s$). Alternatively, the user may wish to specify total battery and monetary budgets in each time period (*e.g.,* $\$, J$), and have the algorithm use dynamic programming to make online decisions. But the multi-stage optimization approach is unlikely to be effective in our setting, as such an optimization often requires knowledge of the distribution of the users' future actions. Predicting users' future actions is remarkably difficult, and is an area we intend to explore in the future based on prior literature in similar domains [12].

3. Time dynamics. Alg. 1 runs periodically, and recomputes a new solution based on current network conditions. For example, if a local model is currently being used, and the network bandwidth improves, DeepDecision may decide to change to offloading. However, if the network bandwidth will only increase temporarily, it may be suboptimal to switch due to cost overhead (*e.g.,* loading a new deep learning model or establishing a new network connection takes time). To reduce frequent oscillations, we analyze the conditions under which a switch should occur, and add this as an outer loop to Alg. 1. We assume there is a throughput predictor that can estimate the future bandwidth and latency over a short period of time [13]. Let T be the length of time of the network conditions change, f_l^*, a_l^* be the optimal solution of Prob. 1 assuming the model decision is fixed to be local, and f_r^*, a_r^*, r_r^* be the optimal solution of Prob. 1 assuming the model decision is fixed to be remote. Due to space constraints, the proof can be found in the technical report [14], and here we present the results directly. DeepDecision should switch from local to remote iff:

$$f_r^* - f_l^* \leq \alpha(a_l^* - a_r^*) \tag{11}$$

And switch from local to remote iff:

$$\alpha(a_l^* - a_r^*) \leq f_r^* \left(1 - \frac{1}{T}\left(\frac{r_r^* f_r^*}{B} + L\right)\right) - f_l^* \tag{12}$$

Note that the conditions are asymmetric because switching from local to remote incurs additional delay while waiting for the first result to arrive from the server, thus decreasing the average frame rate in the objective function (1). Intuitively, the factors that encourage switching from local to remote are: long period of time T of improved bandwidth, high bandwidth B, and low network latency L.

IV. Measurements & Experiments

This section describes our experiments, which serve two purposes. First, we want to understand the interactions between various factors (*e.g.,* processing time, video quality, energy consumption, network condition, the accuracies of different deep learning models) on both the local device and the server. While prior works have carried out limited profiling of running deep learning on servers [15] or on phones [8], to the best of our knowledge, we are the first to explicitly consider the input stream as a video rather than a sequence of images, as well as the impact of network conditions on the offloading strategy, and the tradeoffs of compressible deep learning models. Second, we seek to understand our algorithm's behavior compared to existing algorithms and assess its ability to make decisions on where to perform computation. These baseline comparison algorithms include:

1. Remote-only solution: All frames are offloaded to the backend. Many industrial solutions (*e.g.,* Alexa, Cortana, Google Assistant, Siri, *etc.*) adopt this solution.

2. Local-only solution: All jobs are executed locally. Some specific applications, such as Google Translate, run a compressed deep learning model locally.

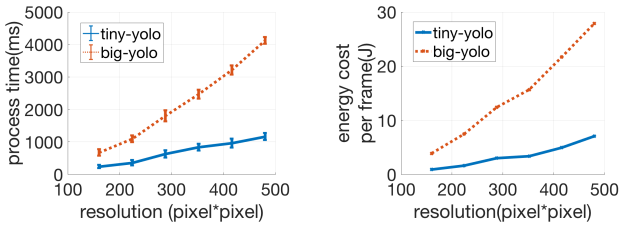
3. Strawman: We implement a "slim" version of MCDNN [8] optimized for the scenario in which the device serves one application. Our strawman picks the model variant with the highest accuracy (defined below) that satisfies the remaining monetary or energy budget. We note that MCDNN does not consider the effects of network bandwidth/latency, how often the neural net should execute, or the impact of delay on accuracy.

A. Testbed Setup

Our backend server is equipped with a quad-core Intel processor running 2.7 GHz with 8 GB of RAM and an NVIDIA GeForce GTX970 graphics card with 4GB of RAM. Our front-end device is a Samsung Galaxy S7 smartphone.[2] We develop an Android version of Yolo based on Android Tensorflow [16] and the Darknet deep learning framework [9]. The Android implementation can run a small deep learning model (called *tiny-yolo*) with 9 convolutional layers, and a bigger deep learning model (called *big-yolo*) with 22 convolutional layers. Both models can detect 20 object classes and are trained on the VOC image dataset [17]. The server runs *big-yolo* only.

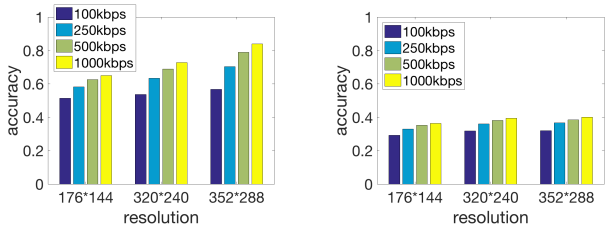
When offloading is chosen by Alg. 1, the front-end device compresses the video frame and chooses the correct frame rate and resolution, then sends the video stream to the server. The stream is sent using RTP running on top of UDP [18]. Videos are compressed using the H.264 codec at one of three target bitrates (100kbps, 500kbps, and 1000kbps). The video frame rate can be set between 2 and 30 frames per second (FPS), and the video resolution can be set to 176×144, 320×240, or 352×288 pixels. Our app also logs the battery usage

[2] We also tested on other smartphones such as the Google Pixel and OnePlus 3T and found similar qualitative behaviors. Our system can work on any front-end device by first running performance characterization offline.



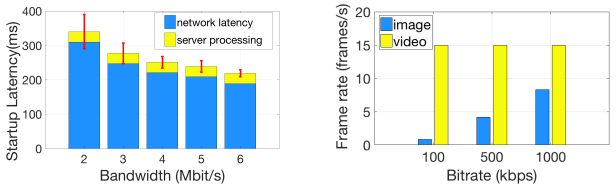
(a) process time vs. resolution (b) energy vs. resolution

Fig. 2: Tradeoffs on the front-end device. (2a): Processing time increases with resolution, especially for *big-yolo*. (2b): Energy usage increases with resolution, especially for *big-yolo*.



(a) Accuracy of *big-yolo*. (b) Accuracy of *tiny-yolo*.

Fig. 3: Model accuracy for different video qualities. Accuracy increases with resolution and bitrate.



(a) Startup latency vs. bandwidth (b) Video vs. image offloadings

Fig. 4: Tradeoffs on the cloud. (4a): Network latency dominates compute latency. (4b): Compressing the offloaded video enables higher frame rates than image offloading.

reported by the Android OS, the data usage, and the time elapsed between sending the frame and receiving the detection result from the server. We feed a standard video dataset [19] to the smartphone to ensure a consistent testing environment for the different algorithms.

Accuracy metric. We measure accuracy using the Intersection over Union (IOU) metric, similar to [5]:

$$\text{IoU} = \frac{area(R \cap P)}{area(R \cup P)} \tag{13}$$

where R and P are the bounding boxes of the ground truth and the model under evaluation, respectively. The average of the object IoUs in the frame gives the frame's IoU. The average of the frame IoUs gives the video's IoU. Our ground-truth model is *big-yolo* executed on raw videos (without any compression) at the 352×288 resolution (we select this particular resolution out of convenience since pre-trained models are available).

B. Measuring tradeoffs without network effects

We first study when the phone runs the deep learning locally, without offloading, to understand baseline performance.

Impact of video resolution: We vary the image resolution from 160×160 to 480×480 pixels. Recall that Yolo can dynamically adjust its internal structure for different resolutions, so its running time is sensitive to the video resolution. In Fig. 2a, we plot the tradeoff between frame resolution and processing time. When the CNN runs on the phone, *big-yolo*'s processing time is between 600ms and 4500ms, whereas *tiny-yolo*'s processing time is between 200ms and 1100ms. Since the processing time increases, we also expect the battery usage to increase. In Fig. 2b, we show the tradeoff between frame resolution and energy consumption per frame. We note that both processing time and energy consumption scale linearly with the width/height of a video. These two functions are used as input to Alg. 1 (specifically, $\ell_i^{\text{CNN}}(p)$ and $b_i(p)$, for $i > 0$). We also measure the energy of offloading, and find its mean value to be 2900 mW, independent of bitrate and frame rate.

Parameterizing accuracy. We next study the correlation between the accuracies of deep learning models under different image qualities. We use two parameters to determine the clarity of a video/image sequence. *1. Resolution:* This is intuitive, because higher resolution often corresponds to better image quality; and *2. Bitrate:* Resolution by itself does not determine the image quality, as the number of bits used to encode that resolution also matters. A low bitrate will cause the video encoder to aggressively compress the frames and create distortion artifacts, decreasing the video quality and the prediction accuracy.

We seek to understand how the video resolution and bitrate interact with the model accuracy. To do this, we encode the videos in different combinations of resolutions and target bitrates, measure their accuracy, and show the results in Fig. 3. Which factor is more important for accuracy, the resolution or the bitrate? We observe that increasing the resolution without increasing the bitrate has a limited impact on the prediction quality; for example, in Fig. 3a the 100kbps bar stays (almost) flat for different resolutions. However, if the bitrate increases along with resolution, there can be substantial accuracy improvement, as shown in Fig. 3a for the high-resolution case.

The non-linear interactions between bitrate, resolution, and accuracy suggest the need for a sophisticated offloading decision module that will carefully consider the complex tradeoffs between the various resources. In Alg. 1, these tradeoffs are captured by the function $a_0(p, r, \ell_0)$.

C. Measuring tradeoffs with network effects

Next, we study the impact of the (communication) network conditions on system performance. The Samsung Galaxy S7 phone is located in the same subnet as the server, and for the sake of these measurements, always chooses to offload. We use the `tc` traffic control tool to emulate different network conditions and use data usage to estimate LTE monetary cost.

End-to-end latency. The end-to-end latency l_i experienced by the viewer is the sum of the processing latency ℓ_i^{CNN} plus the network transmission time. We measure the end-to-end latency of each frame as well as the frame rate, when varying

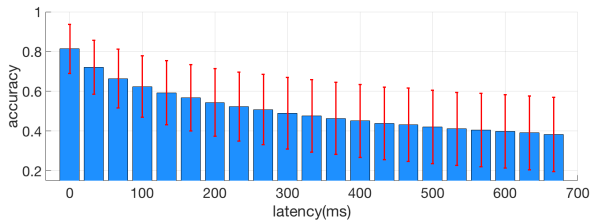
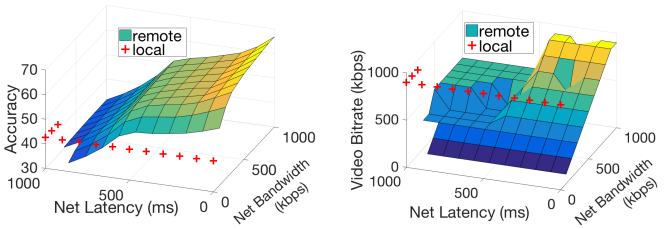


Fig. 5: Accuracy as a function of total latency (model processing time + network transmission time). Accuracy decreases as the latency increases, due to stale frames, especially for high-motion videos.



(a) Accuracy (b) Video bitrate

Fig. 6: Better network conditions result in higher accuracy. To achieve good accuracy when the network latency is large, the bitrate must be carefully chosen to reduce the total transmission time.

the bandwidth and latency between the client and the server. The latency with unconstrained bandwidth between the client and the server is about 30 ms. We repeat each trial 30 times with a frame resolution of 352×288 pixels, and plot the results in Fig. 4a. We observe that the network transmission time consumes the majority of the total latency, while the server's *big-yolo* CNN generally executes in less than 30 ms. This indicates that network transmission is the key driver of latency and frame rate, rather than the CNN processing time, and that any offloading strategy should be highly aware of the current network conditions when making a decision.

Impact of latency on accuracy. Network latencies can cause delayed delivery of the output, decreasing the accuracy. For instance, suppose at time $t = 0$, a frame is sent to the back-end server, processed, and the result returned to the front-end device at $t = 200$ ms. At that time, if the scene captured by the camera has changed (for example due to user mobility), the detected object location, and thus the overlay drawn on the display, may be inaccurate. Hence, any system that performs offloading of real-time scenes must take this delay into account when measuring the accuracy; however, previous works generally compute the accuracy relative to the original time of frame capture [8], [7]. To understand the impact of latency on accuracy, for each video at fixed (resolution, bitrate), we measure how the accuracy changes as a function of latency (i.e., how changing the round-trip time will affect prediction accuracy). A sample result for 1000 kbps, 30 FPS videos at 352×288 resolution is shown in Fig. 5, where the height of the bar is the mean accuracy across videos and the error bar is the standard deviation. We observe that accuracy decays slowly for these particular videos, which have relatively little motion. One qualitative observation we make is that videos with more active subjects (*e.g.,* foreman [19]) tend to have much shorter half-life than videos with "talking heads" (*e.g.,* akiyo [19]). The relationship between accuracy and latency is modeled as $a_0(p, r, \ell_0)$, where ℓ_0 is the latency.

Video compression. Finally, DeepDecision also leverages the benefits of video compression. In contrast to previous works which mainly consider videos as a sequence of images, our system encodes the video as a group-of-pictures with I and P frames, significantly reducing the network bandwidth. To show this, we measure the frame rate of the offloaded scene when the scene is encoded as an image versus as a video. (Specifically, to compute the image frame rate, we divide the network bandwidth by the size of each frame when saved as an independent image.) We plot the results for different target bitrates in Fig. 4b, and observe that encoding the scene as a video can help us send $10\times$ more frames to the backend when the network conditions are poor and the target bitrate is low (100 kbps), and $2\times$ more frames when the network conditions are good and the target bitrate is higher (1000 kbps).

D. Performance evaluation

We now study the behavior of the DeepDecision, and compare its performance against baselines.

Different network conditions. First, we examine how our algorithm's decision changes for different network conditions (latency and bandwidth). See Fig. 6. As the interactions between video quality (free variables), network conditions (constraints), and accuracy (objective) are complex, the decision boundaries formed by our algorithm are non-smooth and sometimes even discontinuous. In Fig. 6a, we plot how the accuracy of the machine learning model chosen by our algorithm changes with network latency and bandwidth. The red dots are scenarios where DeepDecision chooses to execute a model locally, and the plane represents choosing the offload. One can see that when there is no network connectivity, DeepDecision is forced to choose local models. Another scenario where DeepDecision chooses local models is when the bandwidth is non-zero but the latency is very large (1000 ms), because there is too much transmission delay to the server, resulting in stale frames and decreased remote accuracy. Note that sometimes DeepDecision prefers remote models even their accuracies are lower than local models (when latency is slightly less than 1000 ms, and bandwidth is small but non-zero). This is because backend models are faster so we can process more frames, which will increase the objective function (1). In general, when the bandwidth increases and/or the network latencies decreases, the performance of DeepDecision improves.

Fig. 6b shows how DeepDecision chooses the video bitrate when the network conditions change. We note that while the accuracies shown in Fig. 6a are deceptively smooth when we decide to offload, the bitrates chosen by DeepDecision to achieve these accuracies are highly non-smooth (as a function of network conditions) and sometimes discontinuous. In particular, the intuition is that when network latency is

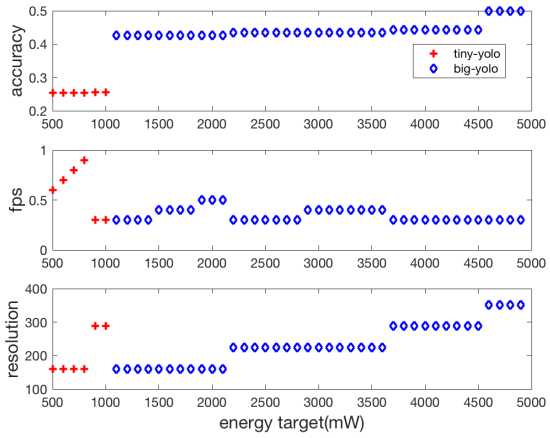


Fig. 7: Impact of energy budget on accuracy and frame rate metrics. With additional energy budget, DeepDecision must choose which metric to increase.

high ($>$ 400 ms), rather than transmitting the video at the maximum possible bitrate, DeepDecision instead (counter-intuitively) offloads at a slightly lower video bitrate in order to save network transmission time and prevent stale frames from decreasing the accuracy of the remote model (Fig. 5).

Energy target. The battery (*i.e.,* energy target) plays an important role when network conditions are poor (namely, very long latency or very low bandwidth). See Fig. 7 for how DeepDecisionmakes decisions under such a harsh circumstance. This figure illustrates three major decision variables as determined by our algorithm: accuracy, frames per second (fps), and resolution. The red dots mean that the DeepDecision decides to run *tiny-yolo* locally. The blue dots mean that DeepDecision decides to execute *big-yolo* locally.

The main observation is that the energy budget needs to exceed a certain threshold in order to start using *big-yolo*. When the phone has an extremely small battery target, it is only able to execute *tiny-yolo*, which uses less energy (see Fig. 2b). If the battery target increases, the question is whether DeepDecision should use that extra energy to (a) increase the frame rate of the current model, (b) increase the accuracy of the current model by increasing the resolution, or (c) bump up the accuracy overall by upgrading to a more powerful model? Our results in Fig. 7 show that initially, DeepDecision will try to increase the frame rate while keeping accuracy unchanged. Then, with more battery, DeepDecision tries to increase resolution to allow the current *tiny-yolo* model to have higher accuracy. Finally, when the battery target is large, DeepDecision chooses the more powerful *big-yolo* model.

Comparison against baselines. In this set of experiments, we evaluate the real-time performance of DeepDecision. In our testbed, we vary the network bandwidth from 0-1000 kbps, allow the network latency to fluctuate naturally, and plot the accuracy over time in Fig. 8. We also plot the performance of the baseline algorithms (strawman, local-only, and remote-only).[3] DeepDecision estimates the network bandwidth and latency by sending small 50 kB probe packets every second. It uses 3727mW and 60.6% CPU usage when executing remotely, and 2060mW with 38.8% when executing remotely.

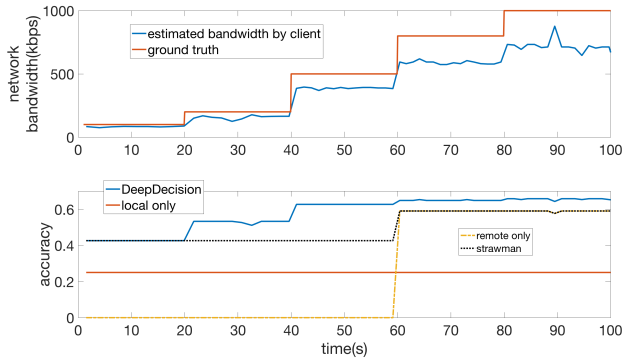


Fig. 8: Performance of DeepDecision compared to baseline approaches. DeepDecision is able to provide higher accuracies under variable network conditions.

Initially, DeepDecision chooses a local model, but as the network bandwidth increases over time, it switches to offloading at around $t = 20$, which boosts accuracy. Past $t = 20$, DeepDecision selects the right combination of bitrate and resolution to further maximize the accuracy. The local-only approach, on the other hand, always has a low accuracy since it uses *tiny-yolo*. The remote-only approach is not able to run initially when the network bandwidth is low. The strawman approach is slightly more intelligent; it starts offloading around $t = 60$ when the network bandwidth is high enough to support the video bitrate, but suffers from reduced accuracy before that. Moreover, since the strawman uses a fixed resolution, it does not know how to select the right combination of resolution and bitrate after it begins offloading and achieves worse accuracy than DeepDecision. Overall, the accuracy of the model chosen by DeepDecision is always higher than that of the baseline approaches. The frame rate is also high (about 15 FPS, capped by the server processing latency, which is not shown). We see that DeepDecision is able to leverage the changing of network conditions and always provide the best accuracy model to the user, by adapting the video bitrate and resolution accordingly, whereas the baseline approaches are less responsive to changing network conditions.

V. Related Work

Deep learning: Recently, applying CNNs to object classification has shown excellent performance [20], [10]. [9] also used CNNs to perform object detection with an emphasis on real-time performance. [15] compares the speed and accuracy tradeoffs of various CNN models. However, none of these works have considered the performance of CNNs on mobile phones. Several works have studied model compression of

[3]Specifically, local-only runs *tiny-yolo* with resolution 160×160, and remote-only runs on the server with a video bitrate of 500 kbps. The strawman is based on [8] and uses a fixed resolution of 320×240 and a video bitrate of 500 kbps, and picks the model (local or remote) with the best accuracy.

CNNs running on mobile phones. [7] uses the GPU to speed up latency, while [21] considers hardware-based approaches for deciding important frames. Our approach is complementary to these in that we can leverage these speedups to local processing, while also considering the option to offload to the edge/cloud.

Mobile offloading: [22] developed a general framework for deciding when to offload, while [23], [24] specifically study interactive visual applications. These frameworks cannot directly be applied to our scenario because they do not take into account that machine learning models may be compressed when executed locally as opposed to remotely. [5], [25], [26] explore remote-only video analytics on the edge/cloud, whereas we focus on client-side decisions of where to compute. Specifically, [5] considered modifying the data (sending a subset of frames) to reduce latency, while we also consider modifying the machine learning model to reduce latency. [25] offloads processing from Google Glass to nearby cloudlets. [26] performs resource profiling similar to our work, but focuses on server-side scheduling whereas we focus on client-side decisions. [27] provides some initial on-device profiling. The closest to our work is perhaps [8], which decides whether to offload CNNs to the cloud; however, they do not consider the current network conditions or profiling of video compression, energy consumption, or machine learning accuracy, which can greatly impact the offloading decision.

VI. Conclusions and Future Work

In this work, we developed a measurement-driven framework, DeepDecision, that chooses where and which deep learning model to run based on application requirements such as accuracy, frame rate, energy, and network data usage. We found that there are various tradeoffs between bitrate, accuracy, battery usage, and data usage, depending on system variables such as model size, offloading decision, video resolution, and network conditions. Our results suggest that DeepDecision can make smart decisions under variable network conditions, in contrast to previous approaches which neglect to tune the video bitrates and resolution and do not consider the impact of latency on accuracy. Future work includes using object tracking to reduce the frequency of running deep learning, generalizing the algorithm for a larger set of edge devices, and customizing the algorithm for different categories of input videos. The hope is that architectures such as DeepDecision will enable exciting real-time AR applications in the near future.

VII. Acknowledgements

Xukan Ran, Haoliang Chen, and Jiasi Chen were in part supported by the Hellman Foundation. The authors thank Amazon for partly providing AWS Cloud Credits for this research.

References

[1] Thomas Olsson et al. Expected user experience of mobile augmented reality services: A user study in the context of shopping centres. *Personal Ubiquitous Comput.*, 17(2):287–304, February 2013.

[2] Michael Irving. Horus wearable helps the blind navigate, remember faces and read books. http://newatlas.com/horus-wearable-blind-assistant/46173/, 2016.

[3] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *IEEE CVPR*, 2014.

[4] Wenxiao Zhang, Bo Han, and Pan Hui. On the networking challenges of mobile augmented reality. *ACM SIGCOMM Workshop on VR/AR Network*, 2017.

[5] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. *ACM SenSys*, 2015.

[6] Loc Nguyen Huynh, Rajesh Krishna Balan, and Youngki Lee. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices. In *ACM WearSys*, 2016.

[7] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. *ACM MobiSys*, 2017.

[8] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *ACM Mobisys*, 2016.

[9] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *IEEE CVPR*, 2017.

[10] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *NIPS*, 2015.

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016.

[12] Diala Naboulsi, Marco Fiore, Stephane Ribot, and Razvan Stanica. Large-scale mobile traffic analysis: a survey. *IEEE Communications Surveys & Tutorials*, 18(1):124–161, 2016.

[13] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. *ACM SIGCOMM*, 2015.

[14] Xukan Ran, Haoliang Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics (technical report). http://www.cs.ucr.edu/~jiasi/pub/deepdecision_infocom17_techreport.pdf.

[15] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *IEEE CVPR*, 2017.

[16] Tensorflow android camera demo. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android, 2017.

[17] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[18] libstreaming. https://github.com/fyhertz/libstreaming, 2017.

[19] xiph.org video test media. https://media.xiph.org/video/derf/, 2017.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[21] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. *ACM MobiSys*, 2017.

[22] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. *ACM MobiSys*, 2010.

[23] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *ACM MobiSys*, 2011.

[24] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *ACM/IEEE Symposium on Edge Computing*, 2017.

[25] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. *ACM MobiSys*, 2014.

[26] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX NSDI*, 2017.

[27] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering deep learning to mobile devices via offloading. *ACM Sigcomm Workshop on VR/AR Network '17*, 2017.