Frugal Following: Power Thrifty Object Detection and Tracking for Mobile Augmented Reality

Kittipat Apicharttrisorn
University of California, Riverside
kapic001@ucr.edu

Xukan Ran
University of California, Riverside
xran001@ucr.edu

Jiasi Chen
University of California, Riverside
jiasi@cs.ucr.edu

Srikanth V. Krishnamurthy
University of California, Riverside
krish@cs.ucr.edu

Amit K. Roy-Chowdhury
University of California, Riverside
amitrc@ece.ucr.edu

ABSTRACT

Accurate tracking of objects in the real world is highly desirable in Augmented Reality (AR) to aid proper placement of virtual objects in a user's view. Deep neural networks (DNNs) yield high precision in detecting and tracking objects, but they are energy-heavy and can thus be prohibitive for deployment on mobile devices. Towards reducing energy drain while maintaining good object tracking precision, we develop a novel software framework called MARLIN. MARLIN only uses a DNN as needed, to detect new objects or recapture objects that significantly change in appearance. It employs lightweight methods in between DNN executions to track the detected objects with high fidelity. We experiment with several baseline DNN models optimized for mobile devices, and via both offline and live object tracking experiments on two different Android phones (one utilizing a mobile GPU), we show that MARLIN compares favorably in terms of accuracy while saving energy significantly. Specifically, we show that MARLIN reduces the energy consumption by up to 73.3% (compared to an approach that executes the best baseline DNN continuously), and improves accuracy by up to 19× (compared to an approach that infrequently executes the same best baseline DNN). Moreover, while in 75% or more cases, MARLIN incurs at most a 7.36% reduction in location accuracy (using the common IOU metric), in more than 46% of the cases, MARLIN even improves the IOU compared to the continuous, best DNN approach.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Mobile Augmented Reality, Energy Efficiency, Object Detection and Tracking, Convolutional Neural Network

SenSys '19, November 10–13, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6950-3/19/11…$15.00
https://doi.org/10.1145/3356250.3360044

ACM Reference Format:
Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V. Krishnamurthy, and Amit K. Roy-Chowdhury. 2019. Frugal Following: Power Thrifty Object Detection and Tracking for Mobile Augmented Reality. In *The 17th ACM Conference on Embedded Networked Sensor Systems (SenSys '19), November 10–13, 2019, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3356250.3360044

1 INTRODUCTION

AR is popular in the market today [44] with potential applications in many fields including training, education, tourism, navigation, and entertainment, among others [12]. In AR, the user's perception of the world is "augmented" by overlaying virtual objects onto a real-world view. These virtual objects provide relevant information to the user and remain fixed with respect to the real world, creating the illusion of seamless integration. Examples of AR apps used today include Pokemon Go, Google Translate, and Snapchat filters.

An important task in the AR processing pipeline is the detection and tracking of the positions of real objects so that virtual annotations can be overlaid accurately on top [14, 35, 42]. For example, in order to guide a firefighter wearing an AR headset, the AR device needs to analyze the camera frame, detect regions of interest in the scene (e.g., victims to be rescued), and place overlays at the right locations on the user display [46]. Commercial AR platforms such as ARCore and ARKit can understand the 3D geometry of the scene and detect surfaces or specific instances of objects (e.g., a specific person), but lack the ability to detect and track complex, non-stationary objects [23, 42].

To track real objects, AR apps can use tracking by detection techniques [57], wherein each camera frame is examined anew to detect and recognize objects of interest; both object locations (*e.g.,* bounding boxes) and class labels are output. Tracking by detection is used, for example, by the open-source ARToolKit [1] to track fiducial markers in the scene. To go beyond this to detect non-fiducial objects in the scene being viewed, one can employ state-of-the-art DNN-based object detectors which yield high object recognition and detection precision (with regards to objects in general). However, a naive plug and play of DNN-based object detection and recognition into a tracking by detection framework will exacerbate the already high battery drain of mobile devices, which is of great concern to mobile users [27]. While the screen, camera, and OS do consume a large portion of the user's battery (3-4 W in our measurements), continuous repeated executions of DNNs (even those models optimized for mobile devices, e.g., [28, 52]) will also consume a major portion (1.7-3 W) of the battery.

Recent works have targeted improving the energy efficiency of DNNs (*e.g.,* by using specialized hardware [29] or via model compression [25]); however, they focus on individual DNN executions on individual input images [30], rather than understanding energy consumption across time, as is needed in AR or other continuous tracking applications. Invoking DNN executions on every captured frame in an AR application will cause high energy expenditure even with such mobile-optimized methods.

In this paper, we ask the question: How can AR apps achieve good object detection and tracking performance and yet consume low energy? To answer this, we make the key observation that while using a DNN is important for detecting new objects, or when significant changes to a scene occur, lightweight incremental tracking can be used to track objects otherwise, in between DNN executions. This saves precious computation and energy resources, but requires initial knowledge of the object to be tracked (which must be supplied by the DNN). To realize such an approach, however, a key question that needs to be answered is "when should DNNs be invoked and when is incremental tracking sufficient to maintain similar accuracies as the DNN?" Although tracking by detection and incremental tracking have been studied together to a limited extent [36, 72], these prior approaches either trigger the DNN at a very high frequency (*e.g.,* every 10 frames), use heavyweight object trackers, and/or assume complete offline knowledge of the video. These limitations make such methods inappropriate for real-time AR applications and/or mobile platforms with battery limitations.

As our main contribution, we design and implement MARLIN (Mobile Augmented Reality using LIghtweight Neural network executions), a framework that addresses the critical problem of limiting energy consumption due to object tracking for AR, while preserving high tracking accuracy. Specifically, MARLIN chooses between DNN-based tracking by detection and incremental tracking techniques to meet three goals: (a) good tracking performance, (b) very low energy drain, and (c) real-time operations. Briefly, MARLIN first performs DNN-based tracking by detection on an initial incoming frame to determine the object locations. Once such objects are detected, MARLIN performs incremental tracking on them to continuously update the locations of the relevant AR overlays; the tracker also checks every frame for significant changes to the object (e.g., a car door opening) to determine if tracking by detection needs to be re-applied. In addition, MARLIN employs a novel change detector that looks for changes to the background (e.g., appearance of new objects) that are likely in the AR scenarios of interest.

MARLIN addresses several challenges in the domain of energy-efficient AR: (1) It provides highly accurate object classification and dynamically tracks the changing locations of multiple different objects in the scene, in order to place the virtual overlays correctly. (2) It reduces CPU throttling in cases where object detection computation demands exceed the compute capability, since built-in CPU throttling can significantly worsen tracking performance; (3) It preserves accuracy while reducing energy in challenging environments such as occlusions and/or zooming which are likely when the AR camera is worn/held by a mobile user; specifically, it does not over-trigger DNNs in response to camera motion; and (4) MARLIN is software-based and does not need specialized hardware. Thus, it is compatible with most modern mobile platforms. MARLIN's software (executables) can be downloaded via the project website [4]. To the best of our knowledge, this is the first detailed design, implementation and evaluation of an energy-thrifty object detection and tracking software framework for mobile AR. Overall, our contributions are as follows:

- We develop a framework, MARLIN, to manage the energy usage of AR, by mediating between two different object tracking approaches: tracking by detection using DNNs and incremental tracking via lightweight methods. MARLIN balances between achieving good tracking accuracy and energy efficiency by triggering DNNs only when needed. The decreased computation demands of MARLIN also reduce instances of automatic CPU throttling and its negative consequences on system performance.
- Within MARLIN, we design a novel lightweight change detector to determine when to trigger DNN detection, with very low false positive rates (crucial for reducing energy usage). Our key idea is to only examine portions of the frame outside of currently tracked objects to determine if new objects are present, while also ignoring effects from camera motion and occlusions.
- We implement and evaluate MARLIN on Android smartphones, using both standard video datasets [37] and through live experiments. Our results show that MARLIN can save energy by up to 73.3%, while losing at most 7.36% accuracy for 75% of the cases as compared to Tiny YOLO, the best baseline periodic DNN-based tracking by detection method we found in our experiments. Surprisingly, we find that in 46.3% of the cases, MARLIN both saves energy and improves accuracy, a win-win situation, compared to this best baseline. This is because MARLIN uses temporal information to avoid triggering tracking by detection, when the scene is noisy and thus detection would likely yield wrong conclusions.
- MARLIN is designed as a general framework that can work with a developer's chosen DNN, with or without a mobile GPU, and still save energy. To illustrate this, we incorporate multiple different DNN models (Tiny YOLO [52], MobileNets [56], MobileNets using mobile GPU [61], and quantized MobileNets [32]) into MARLIN's framework, and show that across these models, MARLIN can save energy by 45.1% while losing 8.3% accuracy, on average (compared to baselines of continuous DNN executions).

2 MOTIVATION

The need for DNNs in emerging AR applications: AR systems are capable of understanding the 3D geometry of the scene (e.g., using simultaneous localization and mapping), but object detection is needed in AR to determine the locations of the virtual annotations in the first place [14, 35, 42]. Current AR systems used in practice are only capable of identifying surfaces or detecting specific instances of objects. For example, the open-source ARToolKit library [1] is designed to track specific fiducial markers placed in the scene (e.g., a QR code), while Google ARCore and Apple ARKit [5, 20] can detect flat surfaces or specific instances of flat objects (e.g., a specific magazine cover, but not the general class of magazines). These object detection capabilities are insufficient for AR applications such as public safety, where general classes of potentially moving, non-flat objects must be detected and recognized with high accuracy (e.g., moving victims needing rescue).

To demonstrate this, we experimented with a demo ARCore app [21] to detect objects of interest (Fig. 1a). We supplied ARCore

(a) ARCore [20] object detection fails for non-affine transformations. **(b) Frequent DNN executions drain battery.**

Figure 1: Detection with ARCore; Energy drain with DNNs.

with an image of a magazine for its internal training. At test time, ARCore was only able to detect the magazine under certain conditions: if the magazine was flat and non-moving. Based on our understanding of the code (full details are unknown because the code is closed source), we hypothesize that this is because ARCore only searches the camera frame for affine transformations training set items (i.e., translation, scaling, shearing, or rotations), and only when the scene is static - a bent object represents a non-affine transformation from a training image, and thus, detection fails.

Such poor or inaccurate detection/classification could result in missing or misplaced virtual overlays, potentially obscuring key portions of the scene and/or confusing the user. Therefore, we argue that the use of state-of-the-art DNNs, which consistently win the ImageNet object detection competition [55], is apt in order to correctly locate and classify the objects in the scene. DNNs are capable of detecting general categories of objects (e.g., human, animal, vehicles) under a variety of conditions, even if that specific object has never been seen before in the training set. For example, later in §5.3, we show that our DNN-based prototype can successfully detect people with high precision, even though we never used their specific images to train the DNN. Compared to classical SIFT features and other machine learning methods from the AR literature [33, 63, 69], DNNs provide more than 2× accuracy improvements [71].

Unfortunately, a naive approach of plugging in DNN object detectors into current AR systems is likely to lead to poor performance due to the uninformed patterns of DNN executions. For example, ARToolKit runs object detection as often as possible (i.e., tracking by detection). Modifying its object detector to call a DNN would result in high energy expenditure due to almost continuous executions. This is true even when using relatively lightweight, compressed DNNs (e.g., Tiny YOLO [52]) optimized for mobile devices (more details later). On the other hand, ARCore and ARKit, to the best of our understanding (the details are closed-source), only record the initial pinned location of an object from when it is first detected, and cannot incrementally track objects while they are moving [21]. Modifying ARCore/ARKit to call a DNN (which may not even be possible due to their closed-source nature) may improve the initial placement of the virtual overlay, but the overlay may not be able to follow moving objects. In our evaluation (specifically Fig. 6 in § 5.2), we show that executing an object detector only once at the beginning of tracking leads to low accuracy.

Energy costs due to frequent DNN executions: To ensure high object detection and tracking accuracy, a naive method is to execute DNNs as often as possible, as is done in several prior works [29, 51, 52]. To showcase the energy drain of such an approach, we tested state-of-the-art object detection and tracking

Features \ System	Liu et al. [42]	Over Lay [33]	ARCore [20]	Glimpse [45]	Deep Mon [30]	Tiny YOLO (Default-DNN) [52]	MARLIN
Energy efficient				✓	✓		✓
No specialized hardware	✓	✓	✓			✓	✓
No offloading			✓		✓	✓	✓
Real-time updates	✓		✓	✓			✓
Copes with CPU throttling							✓
Uses DNN	✓				✓	✓	✓
Localizes objects	✓		✓		✓	✓	✓

Table 1: Comparison of MARLIN and related work

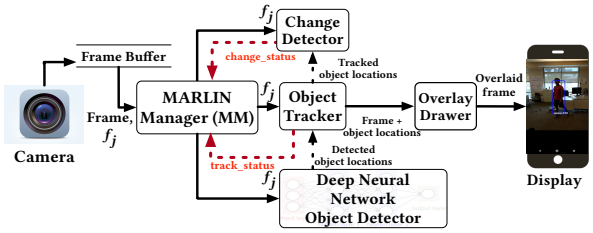


Figure 2: Overview of MARLIN's architecture

on Google TensorFlow, one of the most popular machine learning platforms. We use a popular object detector for mobile devices, Tiny YOLO [52], which applies DNNs as often as possible to maximize the tracking accuracy. This can be expected to result in a rapid depletion of the smartphone battery. To showcase this effect, we perform experiments on a Google Pixel 2, the results of which are shown in Fig. 1b. The rapid energy drain is due to the nature of DNNs, which can contain tens to hundreds of computationally-intensive layers. Furthermore, executing the same model on another recent phone (LG G6) caused the CPU to throttle its duty cycle after the first few minutes of a video, resulting in a significant drop in tracking accuracy (details in §5). We also tested MobileNets on Tensorflow Lite, MobileNets with mobile GPU and quantized MobileNets, and found that this quick battery depletion due to frequent DNN invocations holds true regardless of models or GPU offloading (discussed in §5).

Given the above discussion, we argue that a key gap in realizing object detection and tracking on mobile devices is the lack of a powerful, adaptive, and intelligent framework, designed with the resource limitations on the phone (battery, CPU) in mind. Such a framework should try to achieve a good trade-off between tracking accuracy and energy efficiency. We design and implement such a framework, MARLIN, which is described in the following section. In Table 1, we compare the characteristics of MARLIN with that of other recent AR systems (details in §7).

3 THE MARLIN FRAMEWORK

MARLIN's design is predicated upon the following goals:

- **Low energy:** First, targeted for battery-constrained mobile devices, MARLIN must achieve object tracking with low energy. This not only prolongs battery life, but also saves energy for other AR functions not addressed here (e.g., localization [57]).
- **Real-time performance:** Second, to enable very good AR experience, the detection and tracking of objects of interest must be done in near-real time, i.e., the location of each object must be updated frame to frame (within 33 ms for a 30 FPS camera).
- **Multiple accurate annotations:** Third, since we seek to overlay virtual objects atop the real world, the categories of (multiple)

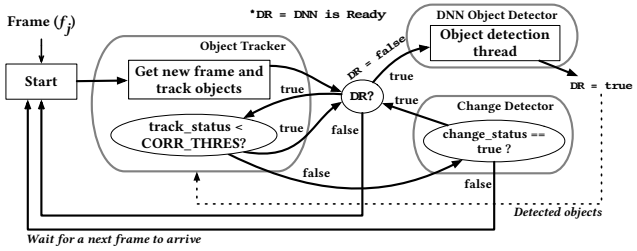


Figure 3: MARLIN **Manager (**MM**)'s decision flow**

real world objects must be classified and their locations must be determined with high precision.

3.1 System Overview

Fig. 2 provides an overview of MARLIN's architecture, composed of pipelined operations from a camera (left) to a display (right). The input to this pipeline is a frame from the camera and the output is a view with overlaid augmented objects (specifically, overlaid bounding boxes in this work) on top of the physical objects (e.g., a person). Each input frame from the camera is buffered before being fetched by the "MARLIN Manager" module. From this point, we abbreviate MARLIN Manager as ***MM***. MM is a real-time scheduler that assigns each incoming frame to one or more of MARLIN's three modules viz., the object tracker, the change detector, and the DNN object detector. These modules act as workers for MM, i.e., each module only processes frames that are assigned to it by MM.

By default, MM assigns a new frame to the *object tracker*, which updates the locations of the objects from one frame to the next. It returns a "track status" which indicates the fidelity of tracking and alerts MM of any changes to the current set of tracked objects, and triggers a new DNN execution if needed.

In addition, to check for new objects in a scene (that require tracking), MM assigns an input frame to the *change detector* module. While many change detection methods exist in the literature (e.g., [3, 73, 74]), we found experimentally that these approaches are unsuitable because they detect changes on both existing and new objects in the scene, resulting in high false positive rates and many unnecessary DNN executions (the main causes being changes due to camera movement or minor changes to the objects being tracked). To tackle this, we designed a new change detector that ignores objects that are already tracked with high accuracy by the object tracker, and only analyzes the portions of the frame that are "external" to the current set of tracked objects. The change detector issues an alert to MM if there are significant changes in these parts.

MM only sends a frame to the *DNN object detector* if it needs to detect/classify new objects in that frame, or when features of the currently tracked objects change significantly and need to be detected anew. This is because the DNN is MARLIN's most energy-draining module and should only be invoked on a need-to basis. MM uses tracking information and the change detection in a principled way to decide if the frame should be assigned to the DNN.

Finally, the object tracker conveys the object locations and the class labels to the *overlay drawer*. The latter draws virtual overlays (bounding boxes) on top of the actual objects in the frame and forwards the augmented frame to the display.

3.2 MARLIN Manager (MM)

In this subsection, we describe MARLIN Manager or MM in greater detail. At a high level, the logic embedded in MM employs the lightweight change detector and object tracker modules as often as possible, and triggers the DNN only if either of these modules indicates that a significant change has occurred in a frame (compared to a prior frame). It uses a "short-circuit OR" decision flow that only runs the change detector if the object tracker did not trigger a DNN, thus avoiding wasted computation/energy.

Functional description: Fig. 3 depicts the decision flow executed by MM . MM obtains input frames from the camera in the form of a byte array with dimensions specified by the three color channels (red, green, blue), and 640×480 pixels (down-sampled from the original resolution, and configurable by the user). Each such frame is assigned to the object tracker. MM waits until the object tracker updates the locations of the objects of interest and returns the correlation between the tracked objects in the current frame and in a previous frame (the returned correlation value is referred to as `track_status`). This correlation captures the fidelity of the tracking across frames (details in §3.3). If `track_status` is less than a threshold (`CORR_THRES`), MM attempts to trigger the DNN. Note here that `CORR_THRES` depends on the desired fidelity of tracking. If higher fidelity tracking is needed, smaller changes (a lower threshold) will need to trigger the DNN (causing these to be more frequent at the cost of higher energy); a lower acceptable fidelity translates to a higher threshold.

If the `track_status` is higher than `CORR_THRES` (meaning that there were no significant changes in tracked objects), then the second operand in the short-circuit OR needs to be evaluated, and so MM starts a change detector thread. This checks if there are changes in the background that could also require the invocation of a DNN. Upon completion, the change detector returns a value (called `change_status`) that indicates whether a significant change in the current frame relative to the immediately preceding frame was detected (details in §3.4). If a significant change is indicated, MM initiates an invocation of the DNN.

In order to prevent repeated DNN invocations due to dynamic changes (e.g., the correlation could be lower for several successive frames), MM checks if or not a DNN invocation has already been made in the immediate past by checking a flag variable, `DR` (for DNN is Ready). If a DNN thread is already being executed, the flag `DR` will be `false` and MM will simply abort the DNN execution attempt. Whenever a DNN is invoked, MM marks the flag `DR` as `false` in order to block other frame assignments to the DNN. Essentially, the `DR` flag ensures that there is only one running DNN thread at any given time, in order to prevent repetitive invocations and thereby ensure that the CPU does not get overloaded or throttled.

Exceptions: If MM cannot finish all the above operations before a new frame arrives, a frame in the buffer is overwritten by a new one. If the change detector thread takes more than one frame to finish (and thus does not return a value within a frame), MM will trigger the DNN at that later time. These exceptions are very rarely observed in our experiments, and even when observed, the delay (2-3 frames) does not affect user experience (not noticeable). If there are no objects being tracked by the object tracker, the tracker returns a zero correlation value, causing a DNN invocation.

3.3 Real-time object tracker

MARLIN needs to continuously track objects of interest (detected by the DNN module) across successive frames, as the object moves/morphs in the scene. To conserve energy, MARLIN's object tracker needs to use (a) very lightweight feature extractors and (b) very lightweight object tracking algorithms. To assess the tracker's performance as it runs, we need some metric that can be computed online; the metric should be able to readily provide a means of determining when the tracking quality has degraded and a new DNN execution is needed (to fully refresh the object locations). We discuss these design considerations and how they influence object tracker design.

Feature extraction: We examined popular feature extractors in the literature. While SIFT features have been used in previous AR systems [33, 34, 69], we chose to use ORB (Oriented FAST and Rotated BRIEF) features in the tracker because they can be extracted in near real-time even on smartphones. ORB has been shown to be 14× and 341× faster than SURF and SIFT respectively with very good tracking precision [54, 67], and we have experimentally verified that extracting SURF/SIFT features for even a single object in a frame takes hundreds of milliseconds, while our object tracker, including ORB feature extraction, takes less than 10 ms (see §5.2).

Object tracking: While heavyweight DNN-based object trackers can provide good tracking accuracy (e.g., [31]), these are unsuitable for mobile devices due to their expensive computation of multiple DNN layers. Our goal here is to estimate the *optical flow* of features, which captures the pattern of motion of objects between successive frames. Instead of trying to design a method from scratch, we use the well-known Lucas-Kanade method [6]. This method estimates the local image flow (velocity) vector (V_x, V_y) using keypoints (features) in the window (in this case the object position box to be tracked) and assumes that these keypoints should move together with this velocity. It has m equations (m keypoints) to solve for two unknowns V_x and V_y, using a least-squares approximation [43]. It makes three assumptions viz., brightness constancy (the same keypoint appearing in both images should look similar), limited motion (keypoints do move very far), and spatial coherence (keypoints move within a small neighborhood) [43]. This method has been shown to be well suited for object tracking [15], and our experiments show that it is also energy-efficient (see Fig. 6 of §5.2).

One important parameter is the neighborhood size that the Lucas-Kanade method searches to find matching features. If the neighborhood size is too small, the object tracker cannot track fast-moving objects accurately. If this neighborhood size is too large, the tracking latency becomes too large because of the larger sample space that needs to be examined for feature matching. We empirically tested this parameter on different videos, measuring the latency and CPU resources utilized for tracking, and found a size of 7 to yield both good accuracy and acceptable latency. A neighborhood size of 7 means that for each feature, the Lucas-Kanade method scans all the features in a 15×15 pixel area to find a matched feature (a center pixel plus 7 pixels above, below, left, and right).

Metric for tracking accuracy: Unfortunately, tracking is not always accurate with respect to changes in object locations. To have a perfect metric to quantify accuracy, we would require the ground truth information about object locations, but this is impossible to have in a real-time, online system. Therefore, in MARLIN, we choose to measure the accuracy of the tracker using the normalized cross-correlation (NCC), which is a well-known technique for template matching [68]. NCC provides a measure of the similarity between two images and is given by: $NCC(f, g) = \frac{1}{|R|} \sum_{i,j \in R} f(i,j) \cdot g(i,j)$ where, f and g are the two images, R is their (bounding box) area, and i, j are the pixel locations within the images.

(a) Frame 1 **(b) Frame 2** **(c) Frame 3**

Figure 4: Cross-correlation decreases from 0.92 (frame 1→2) to 0.69 (frame 2→3) due to occlusion.

Example: Fig. 4 depicts the car in frame 1 to be traced to find its new location in frame 2. The object tracker calculates the NCC between the two boxes by using the above equation, and finds the correlation value is 0.92. Next, the car is tracked from frame 2 to frame 3; the correlation is 0.69 (frame 3 has occluding trees), because of a moderate accuracy drop (i.e., the tracked object is 69% similar to that in the previous frame).

We use a default correlation threshold of 0.3 to trigger the DNN; we consider that if the similarity is less than 30%, the object must be lost (the DNN helps detect objects and recovers accurate locations again). Note that for AR, we need a reasonable level of correlation with respect to the location of a classified object, and "perfect" correlation is not needed. A more stringent threshold (e.g., 0.5) will cause more frequent DNN invocations and thus higher energy. As shown in §5, our default threshold yields good accuracy.

Runtime execution: Putting all of these components together, the object tracker functions as follows. The input to the object tracker is the current frame, and a list of tuples (`objectID`, `classLabel`, `objectLocation`, `detectionConfidence`) containing information about the detected objects. `objectID` is a unique number associated with each detected object, `classLabel` is the class to which the DNN attributes the object (e.g., tiger), `objectLocation` is a 4-tuple vector (left, top, width, height) representing the location of a detected object, and the confidence of the DNN in making the classification decision is given by `detectionConfidence`$\in \{0, 1\}$.

For each detected object, the object tracker executes the following steps: **(i)** For the detected object location in frame j (where j is the most recent frame number seen by object tracker or DNN execution), extract the ORB features F_j (keypoints); **(ii)** For the current frame $j + i$ (i is the number of frames since the last DNN or object tracker execution), extract the ORB features F_{j+i} in the neighborhood of the detected object location from the previous step. **(iii)** Use the Lucas-Kanade method to estimate the optical flow from F_j to F_{j+i} and estimate a new rectangular box that covers the matching features. This new box is the updated location of the object. **(iv)** Compute the minimum NCC (across all objects) between the updated and previous locations (`track_status`) and pass this to MM, which triggers a DNN execution if this NCC is below a threshold.

3.4 Lightweight Change Detector

While the object tracker tracks stable objects and triggers a DNN only when significant changes occur relating to these (i.e., a person's posture changes by quite a bit), MARLIN must also be able to handle new objects that appear in the scene (e.g., a person appears).

To this end, we design a change detector which detects changes *not* pertaining to the objects already being tracked (i.e., new objects coming into view); such changes would also trigger the DNN. The key challenge in designing such a change detector is avoiding high false positives with respect to previously tracked objects (causing extraneous DNN executions). However, our experiments with existing approaches [3, 73, 74] show high false positive rates of approximately 20-100%, resulting in numerous unnecessary DNN executions consuming high energy, even on a simple video with one slowly moving object and a moving camera (detailed results omitted due to space). Towards preventing such false positives, our key idea is to "hide" existing objects from the change detector by changing the corresponding pixels to a common value, whose value does not change across frames.

Functional description: When the change detector receives a frame (and the locations of currently tracked objects) from MM, it converts the frame into a feature vector via the following steps: **(i)** It first colors all rectangular boxes corresponding to the locations of the currently tracked objects white (maximum pixel intensities for red, green and blue channels) to generate what is called a COLORED_IMAGE (example in Fig. 11); **(ii)** It resizes this to 128×128 pixels to form a new image (RESIZED_COLORED_IMAGE), and also calculates the histograms of the red, green, and blue channels of RESIZED_COLORED_IMAGE; **(iii)** Finally, it recasts RESIZED_COLORED_IMAGE, which is a 2D array of pixels, into a single row vector, and appends the three histograms to the end of the row (resulting in another row vector). Thus, it converts an input image of size 640x480x3 (width, height, channels) into a feature vector of size 1x49920 of floating point numbers. This means that we compress it by a factor of 18 (from 921,600 to 49,920 numbers) because we want to quickly perform change detection and do not need all information contained in the frame. Specifically, we focus on the color features and do not use other features such as keypoints, which we experimentally found to be computationally expensive (also shown in [16]).

We reiterate that any changes to tracked objects (now "whited out" in step (i) above) are handled by the object tracker. To detect changes external to these objects, MARLIN uses a random forest classifier with the color features as the input vector. The forest consists of 50 decision trees (total 55,796 nodes). Each (binary) tree has a maximum depth of 20 and each node in the tree is a logical split that takes a variable (an element in the feature vector) and checks its value against a threshold that was learned during model training (details in §5.1). These thresholds represent natural colors of backgrounds (e.g., sky or grass or whited-out pixel) and foregrounds (e.g., tiger or elephant) in order for each node to decide whether or not this frame contains a significant change. The output of each tree is obtained by reaching a leaf node (after moving through splits down the tree) and the final detection result is by a majority vote across all the trees. We also tried other lightweight classifiers such as Support Vector Machines, but found experimentally that random forest had the highest change detection accuracy.

Runtime execution: MM invokes the change detector after the object tracker, which provides the updated objects' locations in the current frame. The change detector then uses the supervised classifier to detect changes to the input feature vector. It inputs the above feature vector to the classifier and outputs 1 (change detected) or 0 (no change detected).

Layer	Filter	Size	Stride	# Params	Layer	Filter	Size	Stride	# Params
c_1	16	3×3	1	448	c_5	256	3×3	1	295,168
m_1		2×2	2		m_5		2×2	2	
c_2	32	3×3	1	4,640	c_6	512	3×3	1	1,180,160
m_2		2×2	2		m_6		2×2	2	
c_3	64	3×3	1	18,496	c_7	1024	3×3	1	4,719,616
m_3		2×2	2		c_8	1024	3×3	1	9,438,208
c_4	128	3×3	1	73,856	c_9	175	1×1	1	179,375
m_4		2×2	2		r				

Table 2: MARLIN's DNN architecture (based on [52]).

Exceptions: In most cases, the change detector reports a change prior to the handling of the subsequent frame. If in the rare case, the change detector finishes its checks after a subsequent frame arrives, the change detection result will be used by MM to trigger the DNN (if needed) as soon as the result is received.

3.5 DNN based Object Detector

Next, we briefly describe the DNN module within MARLIN.

Functional description: The input frame received by the DNN module from MM is passed through 16 layers (using the `recognizeImage()` method of Tensorflow) sequentially as shown in Table 2, where $c_i, i \in \{1, 9\}$ represents a convolutional layer, $m_k, k \in \{1, 6\}$ is a maxpooling layer, and r is a region layer which outputs the final prediction results containing object locations, class labels, and confidence values. The output of c_9 has a dimension of `gridWidth` $\times$ `gridHeight` $\times$ `boxes` $\times$ (`classes` + 5), where `gridWidth` and `gridHeight` are grid dimensions corresponding to the input frame, `boxes` is the number of prediction candidate boxes for each grid cell and `classes` is a list of class probabilities (a value for each class) with respect to object classification. The additional 5 dimensions represent the "objectness" of the predicted box (i.e., the probability that the box contains an object) and the box location `(x,y,w,h)`.

At layer r, a softmax function [7] outputs the confidence that an object belongs to a class. The confidence is computed as `confidence` = `objectness` $\times$ `class_prob`, where `class_prob` is the maximum value from the list of probabilities of belonging to the various classes. If for a given prediction candidate box, `confidence` is less than a threshold, that prediction box is ignored. In our evaluations, we set this threshold as 0.25 because this means that a box will be accepted if `objectness` and `class_prob` are both greater than 0.5. We have empirically found that this threshold yields a reasonable balance between object plausibility and the number of objects detected.

In summary, for each prediction box, the DNN predicts a center point, width, and height of an object, and how likely it is that the box contains an object (`objectness`). It finally outputs the class to which the object in the box most likely belongs (`class_prob`). Tiny YOLO computes these via a single pass through the network (from the image to the prediction), making it one of fastest DNNs for object detection on mobile platforms (latencies of state-of-the-art DNNs are compared in [53]). We also evaluate other possible DNN model choices in §5.2.1. Note that MARLIN executes pre-trained DNNs for real-time inference, with training being performed offline without power constraints (training details provided in §5.1).

Exceptions: If the DNN takes too long to complete, the object tracker has to track incrementally. It is possible that between the time that the DNN receives an input frame i and returns a result in frame $i + j$, there is a significant temporal distance, resulting in the object tracker failing to find the objects in frame $i+j$ detected by the

DNN in frame i. If this happens, MM will invoke the DNN module again until tracking by detection succeeds in finding objects.

4 IMPLEMENTATION

We next briefly describe MARLIN's implementation, which realizes seamless interactions between multiple Android classes/threads.

Platform: We implement MARLIN on Android phones (LG G6 and Google Pixel 2 running Android 7.0 and 8.0, respectively). We use the TensorFlow [58] and OpenCV libraries [8] to implement the DNN and image manipulation functionalities, respectively.

Module implementation: MM runs within a CameraActivity class that extends Activity, the main UI class in Android. It starts when the MARLIN app is invoked by the user. A new frame is buffered in a byte-array in shared memory and MM fetches it once the memory has been written (subsequently the frame is dispensed to the other modules). ***Object Tracker*** is an instance of the class MultiBoxTracker, and provides methods for other components that want to exchange shared information. It runs in the main thread because it is fast (6-10 ms per frame with multiple objects) and does not block the UI. ***Change Detector*** is a background thread that copies a new frame from MM and calls `getTrackedBoxes()` of the object tracker to get the set of currently tracked objects; it also runs the algorithm in §3.4 to detect changes. ***DNN*** is also implemented as a background thread. A DNN thread can be interrupted and can save its intermediate results for further processing when it resumes. This allows the main UI thread to have access to the CPU even when a DNN thread is being run (so that the app is responsive to the user at all times). ***Overlay Drawer*** is a callback thread of the OverlayView Android class and fetches a list of tracked objects from the object tracker and draws them on the frame.

Information sharing: We use methods to pass parameters to/from the object tracker and use shared memory to communicate for real time operations. MM copies a frame to the working threads (change detector or DNN) only if it decides to call one of them.

Frame Synchronization: We use frame sequence numbers to ensure that the different components are synchronized with respect to frames. MM increases the frame sequence number by 1 for each new frame and is the only entity that can update this number.

Logging: MARLIN is instrumented to log CPU frequency, CPU temperature, locations of tracked objects in the scene, and the latency of each component of MARLIN. ***Object location:*** In the object tracker code, we log frame identifiers, object locations, and class labels into storage, and use these logs to compute the accuracy offline. ***Energy:*** Since the phones do not provide direct physical access to the battery, we use software tools to measure energy consumption. On the LG G6, we use Qualcomm's Trepn Power Profiler app [49], and on the Google Pixel 2, we use Android system logs (due to Trepn's lack of support for the Google Pixel 2). Specifically, we read the Android virtual files `current_now` and `voltage_now` from the `/sys/class/power_supply/battery/` directory to obtain current and voltage (used to compute power). The battery level values are read from the `ACTION_BATTERY_CHANGED` Android system variable. ***CPU:*** We read the CPU frequency and temperature from the virtual files `scaling_cur_freq` and `thermal_zone10/temp` every 200 ms. The CPU load is then estimated as $\frac{cpu\_freq}{maximum\_freq} \times 100$. We estimate these metrics because recent Android versions since Marshmallow adjust CPU frequencies in response to load (here mainly DNN executions) in real-time [19].

5 EVALUATIONS

In this section, we describe the experimental evaluations of MARLIN. We first provide brief discussions on details such as our training and test sets and the metrics for evaluations.

5.1 Prerequisites and Metrics

Baselines, Model Training and Inference: We first describe the baselines used for comparisons and the training and test datasets that we use.

Baselines: We consider five different DNN models and perform continuous invocations of these as our baseline cases; we also consider a subset of these models as appropriate as the DNN object detector in MARLIN. The five models are abbreviated as follows: (a) **YOLO** [52], which is a 30-layer DNN detector that provides high accuracy on servers but is typically not used in mobile systems because of its high power consumption and latency; we consider it for completeness but do not use it as an object detector in MARLIN. (b) Tiny YOLO or **TYL**, which is a compressed 16-layer version of YOLO. (c) MobileNets [56] or **MNet**, which is trained and run on the Tensorflow Lite [60] framework. Tensorflow Lite is TensorFlow's lightweight platform for mobile and embedded devices; this provides us with insights with regards to MARLIN's energy savings capabilities on an already optimized mobile software platform. (d) MobileNets using mobile GPU or **MNet-GPU**, which offloads expensive computations to a GPU for low power [30, 39]. (e) MobileNets quantized model or **MNet-Q**, which quantizes the DNN weights in order to reduce execution latencies, and possibly also the DNN execution energy [25, 32].

In terms of notation, when we consider the continuous invocations of one of these DNN models, we include the prefix "Baseline" (e.g. Baseline-TYL). When we use a DNN model as the object detector in MARLIN, we apply the prefix "MARLIN" (e.g. MARLIN-TYL). Because we experimentally find that Tiny YOLO has the best accuracy compared to the other models, we later consider it both as the baseline and as the object detector in MARLIN; thus, we subsequently refer to "Baseline-TYL" as "Default-DNN" and to "MARLIN-TYL" as "MARLIN". Further details are provided in §5.2.

We also compare MARLIN with handcrafted approaches that invoke the Tiny YOLO DNN after skipping a fixed (K) number of frames; the extreme case is when $K = \infty$; *i.e.,* when incremental tracking is used continuously after the initial detection, which we call **Inc. Track**. Our baselines are inspired by similar approaches from the literature (*e.g.,* continuous DNN invocations [30, 51], incremental tracking [57], periodic DNN executions [72]).

Model Training and Inference: In this section, we describe our machine learning model training and testing methodologies.

DNN model training: We train these models with the ImageNet video dataset [55], consisting of 3,862 video clips (1.1 million frames) containing 30 categories of objects, with ground truth labels provided. We split the dataset and use 95% for training and 5% for validation. We calculate model accuracy on the validation set every ten training epochs to check if the model was overfit (accuracy starts to fall). For YOLO models, we adjust learning rates relative

to training epochs as specified in [52], and for MobileNets models we use learning rates specified in the default training scripts [59].

Change detector model training: The change detector is implemented as a random forest classifier trained with 100,000 video frames from the ImageNet dataset. Because the video clips were of different lengths, to avoid biasing the change detector towards longer videos, we randomly chose 30 frames from each video for training. The training set is divided into four subsets: (1) unmodified frames with at least one new object (`change_status` is true); (2) frames with existing tracked objects colored white but with at least one new object in the background (`change_status` is true); (3) frames where all objects in the scene were already tracked and colored white (`change_status` is false); (4) unmodified background frames with nothing else (`change_status` is false). This labeling resulted in 50% of the training set being labelled with `change_status` is true and the other 50% labeled as `change_status` is false.

We experimented with various classifiers (random forest, support vector machines, shallow neural network), and with other input features (e.g. edges, colors, histogram of gradients). On the 10,000-frame validation set, the random forest classifier using color histogram and pixel input features (details in §3.4) achieved the best performance across all tested models, with 88.0% precision and 81.7% recall on the binary classification task. In comparison, e.g., SVM using HOG features has 64.9% precision and 61.4% recall.

Model inference: After training the models offline on a server, we load them on Android phones with the appropriate TensorFlow and OpenCV libraries. While we evaluate the system performance using accuracy and energy metrics (details upcoming), DNN inferences are called by MM. Note that neither the DNN models nor change detector models see the test videos during training time.

Metrics: We evaluate MARLIN 's accuracy in classification and tracking and its energy consumption.

Accuracy metrics: To quantify the accuracy of classification and tracking we use the following metrics [11, 66]:

- *Average Classification Precision (ACP):* Given frame i, we compare the predicted class labels with ground truth labels and count all the matches as *true positives (TP)*. We count unmatched labels as *false positives (FP)*. Then, the ACP of frame i is $ACP_i = \frac{TP}{TP+FP}$. The ACP of a video is computed as the average ACP of its frames.
- *Average Intersection Over Union (IOU):* If the predicted class label of an object matches a ground truth label, we calculate the IOU as the overlap between the predicted and ground truth regions. We perform dataset experiments where we use the provided ground truth data; we also do live experiments where we use a powerful object detection method, viz., YOLO (details in §5.3) as the ground truth. The IOU of object j in frame i is $IOU_j^i = \frac{R_j^G \cap R_j^P}{R_j^G \cup R_j^P}$, where R_j^G is the ground truth region of object j, and R_j^P is the predicted region of object j. We average the IOU for all the predictions per frame, and finally average the IOU across all frames in the video.

We point out that even the state-of-the-art object trackers achieve at best a 65% location accuracy [11] using the IOU metric (for example, a 65% IOU corresponds to 79% of the predicted region overlapping with the ground truth region, if both regions have the same area, using the equation above). These accuracies suffice for the applications we have in mind; the relatively low accuracy only causes small displacements of the real-world objects, and thus does not majorly affect the placement of augmented objects.

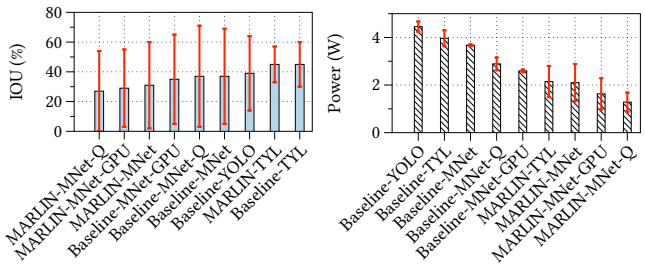


Figure 5: With four different DNN models, MARLIN **saves 45.1% power while losing 8.3% IOU, on average.**

Energy metrics: We use power and battery life to evaluate the energy consumption of MARLIN. We log energy samples every 200 ms (as detailed in §4) and compute the average over the period of an experiment to compute power. To measure the energy of MARLIN's individual components, we successively enable each component and estimate the additional energy consumption as that component's power. For example, if we measure the OS plus screen as consuming 1000 mW, and then enable the camera and measure a total power of 2800 mW, we esimate the camera's power as 1800 mW. To compute battery life, we record the starting battery level (b_s) and the final battery level (b_f) in each experiment (according to §4). We then perform linear regression to estimate the total battery life as $BL = \frac{p \times 100}{(b_f - b_s) \times 60}$, where p is the duration (minutes) of each experiment.

5.2 Offline Dataset Experiments

First, we evaluate MARLIN's performance offline on a standard video dataset with known ground truth, across a diverse set of environments. Our complete dataset includes 80 test videos [37] with a variety of objects (*e.g.*, trains, animals, cars), single and multi-object scenes, and fast and slow-moving scenes, meant to emulate a variety of settings under which AR could be used. In each video, the number of objects varies between 1 and 15, and the average object motion between consecutive frames (the Euclidean distance between an object's center in frames i and $i+1$) ranges from 0.5 to 10.7 pixels. Since the videos are relatively short (hundreds of frames), and we want to capture the effect of a longer AR experience within the same environment, we loop the videos to have a total duration of 10,000 frames per video. We allow a 5-minute cooldown period between each video to reset the phone's state.

To begin with, to keep the time duration of experiments within reason (given the limited number of phones at our disposal), we consider 15 videos and compare the performance of MARLIN with all the baselines and DNN models described earlier, as well as several handcrafted frame skip approaches. Each set of experiments with a given DNN takes three hours (running 15 videos, cool down, phone recharging). These experiments represent different types of object classes and various levels of motion. From § 5.2.3 we present experimental results with the entire set of 80 videos and compare the performance of MARLIN with the best found DNN (Tiny YOLO).

5.2.1 Comparison with the baseline approaches. **Compared to continuous executions of compressed DNNs that are optimized for mobile devices,** MARLIN **reduces power by 45.1% while losing 8.3% IOU, on average.** We plot the average power

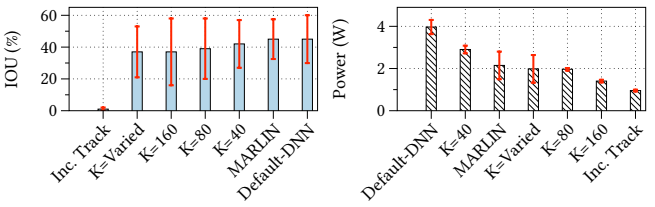


Figure 6: Compared to only tracking or periodic DNN executions, MARLIN has higher accuracy and/or lower energy.

and accuracy of the various approaches considered in terms of IOU in Fig. 5. First, we note that uncompressed YOLO consumes the most power due to its model complexity, but its average IOU over time is lower than Tiny YOLO (its compressed counterpart) due to its high detection latency (4500 ms vs. 1200 ms). This is because when detection latency is high, YOLO fails to detect fast-moving objects (e.g., a landing airplane) in time. Therefore, we focus on compressed and optimized models such as Tiny YOLO and MobileNets.

Second, we note that continuous execution of MobileNets (Baseline-MNet) achieves lower IOU and consumes similar energy to continuous execution of Tiny YOLO (Baseline-TYL) [1]. Third, MARLIN with MobileNets (MARLIN-MNet) saves 42.8% power consumption with a 10.6% reduction in IOU, compared to a continuous execution of MobileNets (Baseline-MNet-GPU). Similar energy savings hold for MARLIN with Tiny YOLO (MARLIN-TYL vs. Baseline-TYL), and for MARLIN with quantized MobileNets (MARLIN-MNet-Q vs. Baseline-MNet-Q). Fourth, with regards to the MobileNets variants, (regular) MobileNets, quantized MobileNets, and MobileNets with GPU achieve similar accuracy; in terms of power, mobile GPU and model quantization save 29.3% and 21.3%, respectively (Baseline-MNet-GPU, Baseline-MNet-Q vs. Baseline-MNet). The key observation is that even though the use of the mobile GPU already saves 29.3% of power, MARLIN can further save an additional 37.1% (on top), with a hit of just 9.9% in terms of IOU (MARLIN-MNet-GPU vs. Baseline-MNet-GPU). Overall, these results suggest that MARLIN is a general framework that is useful across a variety of compressed DNN models, even with a mobile GPU. Because it exhibits the highest accuracy (and similar power consumption to other DNN models), we use Tiny YOLO as the default baseline (default-DNN) and as MARLIN's object detector in all subsequent experiments.

5.2.2 Comparison with other hand-crafted approaches. **MARLIN achieves 19× higher accuracy than the incremental tracking approach, and lower energy for the same accuracy compared to the best constant skip approach.** We compare MARLIN against a constant skip approach (with different skip periodicity $K = 40, 80, 160$) and an incremental tracker baseline ("Inc. Track") in Fig. 6 for 15 different videos, where the average number of frames between DNN invocations by MARLIN ranged from 38 to 833. First, we see that "Inc. Track" suffers from very low accuracy compared to all other approaches (19× lower than MARLIN); this is because when the tracker loses track of objects, there is no recovery from object (re)detection available; thus, we do not consider this approach further. MARLIN achieves comparable IOU with the best constant skip approach ($K = 40$) but consumes 26% less power because it intelligently chooses to trigger fewer DNNs. Moreover, even if we "cheat" by hard-coding the value of K to the average value as chosen by MARLIN for each video (K = Varied), the accuracy of MARLIN is still higher on average because MARLIN chooses when to invoke the DNN, as opposed to fixed periodic executions that ignore the scene content. Finally, default-DNN has the same high accuracy as MARLIN but consumes significantly more energy because it invokes additional unnecessary DNNs.

[1]The standard deviation of the IOU for MobileNets tends to be higher than that of Tiny YOLO because MobileNets sometimes misclassifies objects when they are small or blend in with the background, leading to low IOU. See §6 for further discussion.

	OS + Screen	Camera	Object Tracker	Change Detector	DNN
Power	0.9 - 1.1	1.9 - 2	0.2 - 0.3	<0.1	1.7 - 1.9
Latency	-	-	8 ± 2	4 ± 1	1100 ± 100

Table 3: Power (W) and latency (ms) of MARLIN's components.

5.2.3 A closer look at energy and accuracy. **MARLIN extends the battery life by 1.85× on average with a small accuracy loss.** To see whether MARLIN can achieve good performance across a range of videos, we next evaluate the energy savings with MARLIN across a larger test set of 80 videos, and also examine the associated accuracy penalty compared to the default approach, which runs Tiny YOLO as often as possible. In Fig. 7a, we plot the mean and standard deviation of the ACP and IOU across all frames of all videos. For the same experimental runs, we plot the power and battery life in Fig. 7b. These results show that MARLIN reduces power by up to 73.3% (34.5% on average), and extends battery life by 1.85×, with a small loss in accuracy (< 10%). This is because MARLIN triggers tracking by detection significantly less often.

Beyond averages, we also compute the relative power per video as $\frac{p_d - p_p}{p_d}$, where p_d is default-DNN's power consumption and p_p is MARLIN's power consumption. Fig. 8 shows the CDF across videos, and we see that for 75% of the videos, MARLIN reduces power by at least 19% and extends battery life by at least 13%. Also, in 25% of the cases, MARLIN extends the battery life or reduces power by at least 50%. There are only 10% of cases wherein we do not see energy savings; a closer look reveals that these videos have very complex, high motion scenes; thus, DNN-based detection is necessary almost continuously, and MARLIN behaves similarly to default-DNN.

Finally, Table 3 shows a zoomed out view of the power and latency of each component of MARLIN. The results confirm that MARLIN's non-DNN components are lightweight, and focusing on the DNN executions which comprise a large portion of the total energy is key to reducing the overall power consumption.

For 75% of the videos, MARLIN results in at most a 7.3% hit in ACP and a 18% hit in IOU. To understand the performance of MARLIN further, we calculate the relative accuracy of object detection and tracking across videos when using MARLIN and default-DNN (calculation similar to relative energy). The CDFs of relative accuracy in terms of ACP and IOU, across the videos in the test set, are shown in Fig. 7c and 7d. For 75% of the videos, MARLIN results in a hit of ≤ 7.3% (ACP) and ≤ 18.0% (IOU). These modest drops show that MARLIN performs well while ensuring low power in tracking object locations and labels between frames. We note that approximately half of the tested videos are challenging due to fast motion or multiple objects, thus making this result very promising.

Surprisingly, for 46.3% of the videos, MARLIN both achieves better ACP *and* consumes less energy. We see from Fig. 7c and 7d that for a significant fraction of the test videos, MARLIN improves accuracy compared to default-DNN. A closer look indicates that

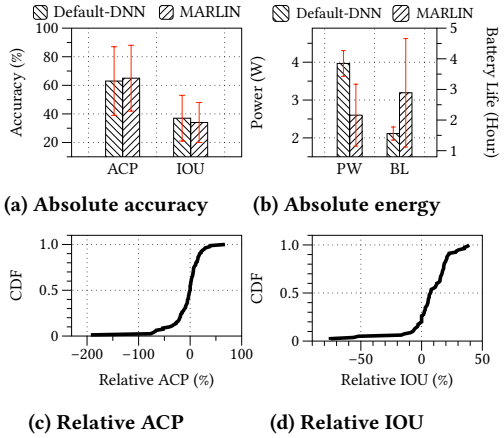


Figure 7: MARLIN **saves energy with mimimal accuracy degradation.**

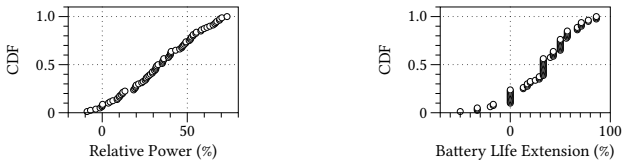


Figure 8: Per video, MARLIN **uses less power (left) and extends battery life (right), relative to the "default-DNN".**

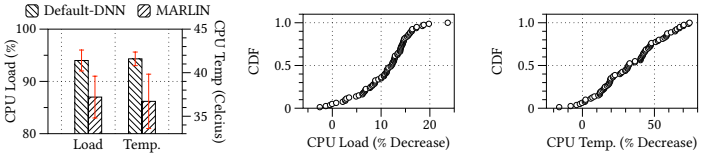


Figure 9: MARLIN **reduces CPU load and temperature (left), relative to the "default-DNN" (center, right).**

for 46.3% of the videos, MARLIN both reduced energy and resulted in higher ACP compared to default-DNN. We find that these cases typically related to videos with a zooming or shaky camera. We will further discuss these special cases next in §5.2.4.

5.2.4 Sample Case Studies. We next present two sample case studies to provide an understanding of why MARLIN sometimes improves accuracy in addition to saving energy; other such cases typically relate to zoomed in frames, occlusions, or cluttered scenes where by using tracking or change detector features, MARLIN reduces DNN invocations that cause false positives/wrong detection.

In the case study of a zoomed-in video, MARLIN **has a 55% gain in ACP and saves 2,500 mW in power.** In this video, the camera is zoomed in on a hamster. In the top two rows in Fig. 10, we plot the IOU over time for default-DNN and MARLIN. We see that default-DNN maintains a reasonable IOU by executing tracking by detection frequently (the dense vertical purple lines), while MARLIN actually *improves* IOU over time. This is because MARLIN 's incremental tracking and change detection use the manually-chosen ORB and color features that are stable over time. Thus, DNNs are hardly invoked. The stability of these features is seen in the bottom two plots in Fig. 10; we show the Euclidean distances between color feature vectors across frames (used by the change detector) and the Hamming distances between ORB feature descriptors between consecutive frames (used by the object tracker).

In contrast, default-DNN chooses features automatically and frequently (with hidden convolutional layers), ignoring temporal correlation and causing the IOU to suffer[2]. More importantly, it yields false positives with respect to detected objects on many invocations. To illustrate this, consider Fig. 11. At frame 1253, both default-DNN and MARLIN detect the hamster correctly in the middle of the frame. The former then triggers the DNN again, which returns two objects in frame 1272: a hamster (true positive) and a dog (false positive) at the right bottom corner. MARLIN, however, continues to track the hamster found in frame 1253 and does not cause an erroneous DNN result in frame 1272. In frame 1272, MARLIN's precision is 100% while default-DNN's precision drops to 50%. We find that this effect repeats for this video and thus, while default-DNN only achieves an overall average ACP of 57% and an IOU of 54% with 400 DNN executions, MARLIN achieves an overall ACP of 87% and IOU of 69%, with only 12 DNN executions. This saves 2500 mW of power and extends the battery life by 3.5 hours.

In the case of a shaky video, MARLIN **improves the IOU by 52%.** An elephant is the focal point of this video, but it is sometimes occluded and suffers from the shaky motion of the camera. We find that only about half of the frames serve as good inputs to the DNN module. Both default-DNN and MARLIN have lower IOUs due to the challenging scene, but MARLIN achieves a 35% IOU while default-DNN only achieves 23%. This is because MARLIN's incremental tracking ignores moderate noises in the scene (e.g., blurry/partially occluded frames), while default-DNN often performs DNN-based detection on such frames and captures poor object features for tracking. For example in Fig. 11, at frame 1729 with both methods, the DNN detects the elephant and outputs a box centered on the elephant and covering most of the body. However, at frame 1748, default-DNN triggers the DNN again but now the center of the elephant is falsely identified to be near the tail. This causes the prediction box to shrink, and the IOU is thus only 40%. MARLIN, on the other hand, does not trigger the DNN since its incremental tracking outputs a more accurate box with an 83% IOU, and the whiting out of the elephant also does not trigger the change detector.

5.2.5 Impacts on Mobile CPU. **For 60% of the videos,** MARLIN **reduces the load and temperature by 10% and 26% or more, respectively.** We measure the CPU load and temperature with MARLIN and compare these to those with default-DNN. Lower CPU load leaves more computational resources for other AR tasks (e.g., pose estimation, lighting estimation), and a lower CPU temperature means a more comfortable user experience when holding/wearing the AR device. Fig. 9 (center and right) shows that in 60% of the cases, the CPU load and temperature are reduced by at least 10% and 26%, respectively (averaged across all 8 cores of the Google Pixel 2 phone). Despite the CPU's cooling technology and operation in a temperature-controlled 20°C room, MARLIN reduces the CPU temperature by 4.88° on average (Fig. 9 left).

MARLIN **significantly helps in coping with CPU frequency throttling.** Automatic CPU throttling lowers the CPU frequency

[2]DNNs that use temporal structure of videos have only been recently studied, e.g., for activity recognition [10] or object tracking [31], and are more complex/high energy [9].

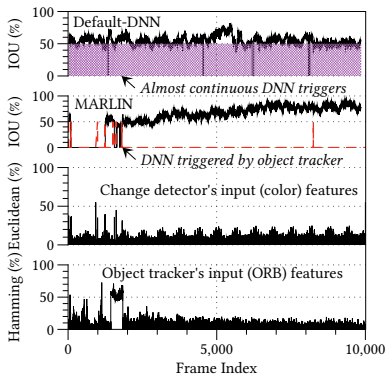


Figure 10: Case study. MARLIN **achieves higher IOU using incremental tracking, rarely invoking DNNs due to the color/ORB features' stability.**

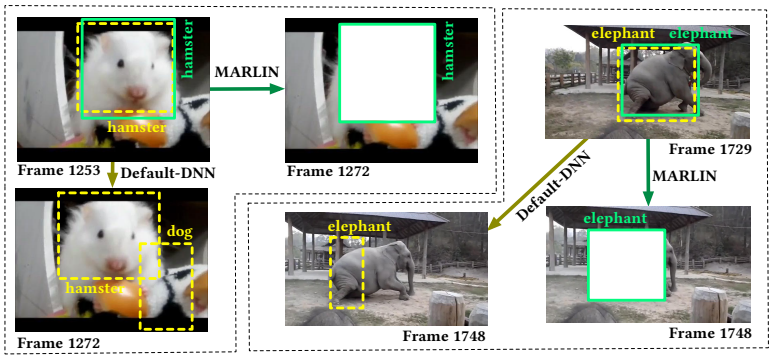


Figure 11: Sample frames of 2 case studies. MARLIN **(solid green) is robust to small variations of currently tracked objects, while default-DNN (dashed yellow) re-triggers the DNN resulting in poor detection.**

based on the load to help conserve energy and reduce the temperature of the chip, and is enabled by default on recent smartphones. While we did not observe CPU throttling on the Google Pixel 2 phone (due to several optimizations [2, 38]), we investigate how MARLIN performs when compared with default-DNN on older processors. Our goal is not to reduce throttling on mobile devices in general, for which methods exist (e.g., [47]), but rather to reduce throttling in the context of object detection and tracking, especially on less powerful mobile devices. Towards this, we next perform experiments on the LG G6, which has a slightly older processor (Qualcomm Snapdragon 821). On this phone, we see that all 4 CPUs work at full speed when executing the DNN, and are automatically throttled after a few minutes of execution. The CPU frequency drops from 1.6 to 1.06 GHz on the two little cores and from 2.35 to 1.06 GHz on the two big cores [41]. Because of this, the power consumption is reduced for default-DNN as shown in Fig. 12b, but MARLIN further improves energy efficiency on the CPU-throttled phone (more power reduction).

Interestingly, we find that CPU throttling causes a 2× increase in the DNN execution latency (taking 1221-2553 ms to execute) and a 80% increase in the object tracker's execution latency (taking 24 ms-43 ms). Thus, DNN-based detection fails more frequently because the scene has already changed by the time the result is returned, especially in moderate to fast motion videos. Figs. 12a and 12b depict the significant decrease in accuracies as compared to a non-CPU-throttled phone; specifically, default-DNN takes a hit of 49.2% in ACP and 54.0% in IOU when throttled. MARLIN triggers the DNN less often, reducing the frequency of CPU throttling, and this improves the accuracies on average. We see this when we compare the relative accuracies of default-DNN and MARLIN on the CPU-throttled phone: for 80% of the videos, MARLIN has a higher ACP and IOU, by an average of 44.0% and 38.7%, respectively (Fig. 12c).

5.3 Live Experiments

To showcase MARLIN's proof-of-concept prototype and evaluate its real-time performance, we perform live experiments in our lab. We train the object detector to detect and overlay virtual objects on people, using VOC2007, VOC2012 [17], and Penn-Fudan Pedestrian [64] datasets for training. We load the trained DNN onto two identical phones (Google Pixel 2), configuring one to run default-DNN and the other, MARLIN. One person holds the two cameras side-by-side, and we request a few student volunteers (2-3) to appear in front of the cameras and act as specified in the scripts shown in Table 5 and a screenshot is shown in Fig. 13. Each trial lasts 30 minutes and the process was approved by our institution's IRB.

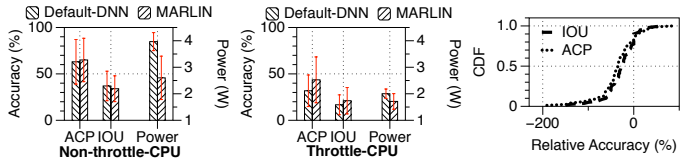
Since we do not have ground truth for these live experiments, we use a more powerful DNN-based tracking by detection algorithm (YOLO [52]) to analyze the video offline on a 12-core Intel Xeon server with 32 GB of memory, and generate annotations considered as ground truth. We also visually inspect a subset of the results to confirm that this is in fact the ground truth.

In live experiments, MARLIN **uses only 18% of power consumed by default-DNN with negligible loss in accuracy, running at 29-30 frames per second.** 30 frames per second is considered good real-time performance for object tracking [14]. Table 4 compares MARLIN's performance with that of default-DNN. In both trials, MARLIN achieves comparable accuracy to that of default-DNN while significantly saving energy. Note here that when measuring the energy, we are careful to remove the consumption caused by auxiliary factors (e.g., the screen and the camera), which are common to both default-DNN and MARLIN. In the first trial, MARLIN uses only 18% of the power compared to default-DNN, and in the second trial, MARLIN uses 51% of the power. The second trial consumes more energy because the human subjects in that trial were slightly more active (more motion). Both MARLIN and default-DNN achieve comparable accuracy in terms of ACP and IOU.

Downloadable software: Our software is downloadable from the project website [4] and tested on smartphones. Both MARLIN and default-DNN methods are provided to enable a relative comparison between the two approaches. Note that when testing with much older phones, they may heat up and cause CPU throttling, impacting both schemes.

6 DISCUSSIONS

Classification accuracy: If the DNN is not trained sufficiently and does not achieve high classification accuracy, this may result in mis-labeling of objects in the scene, and cause the object tracker to either (a) track the wrong objects, or (b) track the right objects but with the wrong label (e.g., track a sheep which is mis-labeled as a horse). Quantitatively, this will manifest itself as low average IOU, since having the correct object label is necessary for a non-zero IOU (see the IOU definition in §5.1). We have observed such



(a) Google Pixel 2 (b) LG G6 (c) Relative accuracy

Figure 12: On a phone with automatic CPU throttling, MARLIN **improves accuracy compared to default-DNN.**

Figure 13: Screenshot

Minute	Live 1	Live 2
0-5	P1 stands with minor movements	P1 sits and P2 stands
5-10	P2 enters and stands casually	P3 enters and walks in random directions
10-15	P1 and P2 walk criss-cross	P1,P2,P3 walk in and out of the camera's field of view
15-20	P1 leaves; P2 walks in random directions	Camera moves towards and away from P1,P2,P3
20-25	P2 returns; P1,P2 walk in random directions	P1,P2,P3 walk in random directions within the camera's field of view
25-30	P2 leaves; P1 walks in random directions	P1 leaves; P2,P3 walk in random directions as the camera moves

Table 5: Live experiment action scripts. P1, P2, P3 are volunteers.

scenarios in initial experiments (later corrected) when Tiny YOLO was not trained for a sufficient number of epochs, resulting in low classification accuracy, and causing MARLIN's object tracker to track the wrong objects. In future work, we plan to further investigate the relationship between classification accuracy and MARLIN's performance, and distinguish between cases where IOU is low due to poor classification or object localization.

Latency of detecting new objects in the scene: When new objects enter the scene (e.g., a person enters the room), MARLIN's change detector (Sec. 3.4) is responsible for detecting that change and triggering a new DNN execution. Since MARLIN uses Tiny YOLO (or other compressed DNNs) as a key component of the system, its performance cannot exceed that of the compressed DNNs in use today; in other words, it cannot detect objects that its constituent DNNs cannot, or even for detected objects, the detection latency cannot be less than that of Tiny YOLO. Qualitatively in our live experiments, we have observed this limitation with both MARLIN and with the baseline Tiny YOLO with continuous execution. However, as researchers develop new DNN models with reduced latencies, MARLIN will automatically be able to leverage these advances by swapping in new, improved DNN models into MARLIN's framework.

7 RELATED WORK

Mobile deep learning: MCDNN [26] chooses which DNN to run given accuracy, latency, and energy requirements of the mobile application. Other efforts speed up DNN inference (e.g., quantized models [25], IDK cascades [65], DeepMon [30]), but only focus on detection and not the use of tracking to reduce DNN invocations. Recent works in computer vision [36, 72] combine detection and tracking, but use expensive DNN-based tracking, frequent fixed interval DNN executions, or offline knowledge of entire video clips. In contrast, MARLIN runs in real-time and adapts DNN executions based on the scene content.

	Method	Accuracy		Energy Consumption	
		ACP (%)	IOU (%)	Battery drop (%)	Power (mW)
Live 1	Default-DNN	90	61	11	1724.55
	MARLIN	92	61	3	319.54
Live 2	Default-DNN	80	56	11	1710.49
	MARLIN	87	51	5	880.65

Table 4: In live experiments, MARLIN **saves significant energy with similar accuracy to default-DNN.**

Mobile AR: Liu et al. [42], Gabriel [24], and Glimpse [14] have proposed cloud/edge-based AR, among others [24, 34, 51, 69, 70]. In contrast, MARLIN focuses on energy efficiency when AR processing is run locally on the device without offloading. Further, Liu et al. [42] focus on partitioned DNN executions on an edge server, by modifying the video encoding parameters, whereas MARLIN considers local execution without video encoding. MARVEL [13] studies energy-efficient AR, assuming the location of the objects in the environment are pre-annotated, while MARLIN studies how to detect and track these objects in the first place. ARCore, ARKit, and ARToolKit [1, 5, 20] provide less sophisticated object detection for planar, non-moving objects, while Vuforia [62] can detect and track up to 20 specific instances of 3D objects. Wagner et al. [63] combine object detection and incremental tracking, but can only detect a single object in the frame.

Change detection: Using the sum of absolute differences is a naive method of change detection, and is susceptible to noise from illumination or background changes [3, 50]. Background/foreground subtraction methods using GMM [73] and KNN [74] are more robust, but assume static cameras, which is not true for AR. Alternatively one could use object detection to check if there are changes over time (e.g. [18]); however, the feature extraction step of such methods are heavy-weight and unsuitable for mobile devices.

Hardware acceleration: There are methods that use specialized hardware sensors to either perform change detection [45] or to tune the energy usage [40]. Qualcomm and Google are developing proprietary chips for computer vision [22, 48]. Such advances are complementary to MARLIN.

8 CONCLUSIONS

Energy consumption is a major concern for AR. We design MARLIN, a framework to reduce the energy consumption of object detection and tracking, which are important in the AR computational pipeline. MARLIN intelligently alternates between DNN object detection and lightweight incremental tracking to achieve high accuracy while saving energy. Our Android prototype shows that MARLIN drastically reduces energy consumption (up to 73% savings) with a minor accuracy penalty (at most 7% for 75% of the test videos), and surprisingly, in 46.3% of the cases, improves both accuracy and energy compared to a default system using DNNs continuously. Future work includes incorporating inertial odometry to further reduce energy consumption.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd for their valuable comments, from which this paper greatly benefited. We also thank the volunteers who participated in our user study. This work has been supported in part by NSF grants 1544969, 1320148, & 1817216.

REFERENCES

[1] [n.d.]. ARToolKit. http://www.hitl.washington.edu/artoolkit/.

[2] Fuad Abazovic. 2017. Qualcomm Snapdragon 835 does not throttle. https://www.fudzilla.com/reviews/43194-qualcomm-snapdragon-835-does-not-throttle.

[3] D Stalin Alex and Amitabh Wahi. 2014. BSFD: Background Subtraction Frame Difference Algorithm for Moving Object Detection and Extraction. *Journal of Theoretical & Applied Information Technology* 60, 3 (2014).

[4] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury. 2019. MARLIN Demo Site. https://sites.google.com/view/marlin-ar/home.

[5] Apple. 2019. ARKit - Apple Developer. https://developer.apple.com/arkit/.

[6] John L Barron, David J Fleet, and Steven S Beauchemin. 1994. Performance of optical flow techniques. *International journal of computer vision* 12, 1 (1994), 43–77.

[7] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[8] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[9] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. 2017. MobiRNN: Efficient recurrent neural network execution on mobile GPU. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*. ACM.

[10] Joao Carreira and Andrew Zisserman. 2017. Quo vadis, action recognition? a new model and the kinetics dataset. In *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 4724–4733.

[11] L. Cehovin, A. Leonardis, and M. Kristan. 2016. Visual Object Tracking Performance Measures Revisited. *IEEE Transactions on Image Processing* 25, 3 (March 2016), 1261–1274. https://doi.org/10.1109/TIP.2016.2520370

[12] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. 2017. Mobile augmented reality survey: From where we are to where we go. *IEEE Access* 5 (2017), 6917–6950.

[13] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E Culler, and Randy H Katz. 2018. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. In *Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 292–304.

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

[15] L. Dan, J. Dai-Hong, B. Rong, S. Jin-Ping, Z. Wen-Jing, and W. Chao. 2017. Moving object tracking method based on improved lucas-kanade sparse optical flow algorithm. In *2017 International Smart Cities Conference (ISC2)*. 1–5. https://doi.org/10.1109/ISC2.2017.8090850

[16] Tuan Dao, Amit K Roy-Chowdhury, Harsha V Madhyastha, Srikanth V Krishnamurthy, and Tom La Porta. 2017. Managing redundant content in bandwidth constrained wireless networks. *IEEE/ACM Transactions on Networking (TON)* 25, 2 (2017), 988–1003.

[17] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88, 2 (01 Jun 2010), 303–338. https://doi.org/10.1007/s11263-009-0275-4

[18] G. Gan and J. Cheng. 2011. Pedestrian Detection Based on HOG-LBP Feature. In *International Conference on Computational Intelligence and Security*. 1184–1187. https://doi.org/10.1109/CIS.2011.262

[19] Google. 2015. Android Marshmallow 3.10 Scheduler. https://android.googlesource.com/kernel/msm/+/android-msm-bullhead-3.10-marshmallow-dr/Documentation/scheduler/sched-hmp.txt.

[20] Google. 2019. ARCore Overview. https://developers.google.com/ar/discover/.

[21] Google. 2019. AugmentedImageActivity.java. https://github.com/google-ar/arcore-android-sdk/blob/master/samples/augmented_image_java/app/src/main/java/com/google/ar/core/examples/java/augmentedimage/AugmentedImageActivity.java.

[22] Google. 2019. Google Cloud TPU. https://cloud.google.com/tpu/.

[23] Google. 2019. Recognize and augment images. https://developers.google.com/ar/develop/unity/augmented-images/.

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

[25] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[26] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. *ACM Mobisys* (2016).

[27] Sean Hollister and Rebecca Fleenor. 2016. How Pokemon Go affects your phone's battery life and data. https://www.cnet.com/how-to/pokemon-go-battery-test-data-usage/.

[28] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[29] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 82–95.

[30] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. *ACM MobiSys* (2017).

[31] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. 2017. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *IEEE conference on computer vision and pattern recognition (CVPR)*, Vol. 2. 6.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[33] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2015. Overlay: Practical mobile augmented reality. *ACM MobiSys* (2015).

[34] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2016. Low Bandwidth Offload for Mobile AR. *ACM CoNEXT* (2016).

[35] Amit Jindal, Andrew Tulloch, Ben Sharma, Bram Wasti, Fei Yang, Georgia Gkioxari, Jaeyoun Kim, Jason Harrison, Jerry Zhang, Kaiming He, Orion Reblitz-Richardson, Peizhao Zhang, Peter Vajda, Piotr Dollar, Pradheep Elango, Priyam Chatterjee, Rahul Nallamothu, Ross Girshick, Sam Tsai, Su Xue, Vincent Cheung, Yanghan Wang, Yangqing Jia, and Zijian He. 2018. Enabling full body AR with Mask R-CNN2Go. https://research.fb.com/enabling-full-body-ar-with-mask-r-cnn2go/.

[36] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. 2016. Object detection from video tubelets with convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 817–825.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[38] Kashish Kumawat. 2017. Snapdragon 835 Review and Benchmark Test. https://www.techcenturion.com/snapdragon-835.

[39] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-power Deep Learning Inference on Mobile Devices. In *International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE Press, Piscataway, NJ, USA, Article 23, 12 pages. http://dl.acm.org/citation.cfm?id=2959355.2959378

[40] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. 2013. Energy characterization and optimization of image sensing toward continuous mobile vision. In *International conference on Mobile systems, applications, and services (MobiSys)*. ACM, 69–82.

[41] I. Lin, B. Jeff, and I. Rickard. 2016. ARM platform for performance and power efficiency - Hardware and software perspectives. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 1–5. https://doi.org/10.1109/VLSI-DAT.2016.7482541

[42] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. *ACM MobiCom* (2019).

[43] Bruce D Lucas and Takeo Kanade. 1981. An iterative image registration technique with an application to stereo vision. *IJCAI* (1981).

[44] Tim Merel. 2017. The reality of VR/AR growth. https://techcrunch.com/2017/01/11/the-reality-of-vrar-growth/.

[45] Saman Naderiparizi, Pengyu Zhang, Matthai Philipose, Bodhi Priyantha, Jie Liu, and Deepak Ganesan. 2017. Glimpse: A Programmable Early-Discard Camera Architecture for Continuous Mobile Vision. *ACM MobiSys* (2017).

[46] Greg Nichols. 2018. https://www.zdnet.com/article/california-firefighters-use-augmented-reality-in-battle-against-record-breaking-infernos/.

[47] Qualcomm. 2019. Snapdragon Power Optimization SDK. https://developer.qualcomm.com/software/snapdragon-power-optimization-sdk.

[48] Qualcomm. 2019. Snapdragon XR1 platform. https://www.qualcomm.com/products/snapdragon-xr1-platform.

[49] Qualcomm. 2019. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler.

[50] R. J. Radke, S. Andra, O. Al-Kofahi, and B. Roysam. 2005. Image Change Detection Algorithms: A Systematic Survey. *Trans. Img. Proc.* 14, 3 (March 2005), 294–307. https://doi.org/10.1109/TIP.2004.838698

[51] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. 2018. DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics. *IEEE INFOCOM* (2018).

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

[53] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[54] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An Efficient Alternative to SIFT or SURF. In *International Conference on Computer Vision (ICCV)*. IEEE, 2564–2571. https://doi.org/10.1109/ICCV.2011.6126544

[55] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[56] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[57] Dieter Schmalstieg and Tobias Hollerer. 2016. *Augmented reality: principles and practice*. Addison-Wesley Professional.

[58] TensorFlow.org. [n.d.]. TensorFlow Android Camera Demo. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android.

[59] TensorFlow.org. 2018. SSDLite MobileNets V2 Config File. https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssdlite_mobilenet_v2_coco.config.

[60] TensorFlow.org. 2019. TensorFlow Lite. https://www.tensorflow.org/lite.

[61] TensorFlow.org. 2019. TensorFlow Lite GPU delegate. https://www.tensorflow.org/lite/performance/gpu.

[62] Vuforia. 2019. Augmented Reality for the Industrial Enterprise. https://www.vuforia.com/.

[63] Daniel Wagner, Gerhard Reitmayr, Alessandro Mulloni, Tom Drummond, and Dieter Schmalstieg. 2010. Real-time detection and tracking for augmented reality on mobile phones. *IEEE transactions on visualization and computer graphics* 16, 3 (2010), 355–368.

[64] Liming Wang, Jianbo Shi, Gang Song, and I-Fan Shen. 2007. Object Detection Combining Recognition and Segmentation. In *Asian Conference on Computer Vision (ACCV)*. Springer-Verlag, Berlin, Heidelberg, 189–199. http://dl.acm.org/citation.cfm?id=1775614.1775636

[65] Xin Wang, Yujia Luo, Daniel Crankshaw, Alexey Tumanov, and Joseph E. Gonzalez. 2017. IDK Cascades: Fast Deep Learning by Learning not to Overthink. *CoRR* abs/1706.00885 (2017). arXiv:1706.00885 http://arxiv.org/abs/1706.00885

[66] X. Wang, M. Yang, S. Zhu, and Y. Lin. 2013. Regionlets for Generic Object Detection. In *IEEE International Conference on Computer Vision (ICCV)*. 17–24. https://doi.org/10.1109/ICCV.2013.10

[67] S. Wu, Y. Fan, S. Zheng, and H. Yang. 2012. Object tracking based on ORB and temporal-spacial constraint. In *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*. 597–600. https://doi.org/10.1109/ICACI.2012.6463235

[68] Z. Yang. 2010. Fast Template Matching Based on Normalized Cross Correlation with Centroid Bounding. In *International Conference on Measuring Technology and Mechatronics Automation*, Vol. 2. 224–227. https://doi.org/10.1109/ICMTMA.2010.419

[69] Wenxiao Zhang, Bo Han, and Pan Hui. 2018. Jaguar: Low Latency Mobile Augmented Reality with Flexible Tracking. In *International Conference on Multimedia*. ACM, 355–363.

[70] Wenxiao Zhang, Bo Han, Pan Hui, Vijay Gopalakrishnan, Eric Zavesky, and Feng Qian. 2018. CARS: Collaborative Augmented Reality for Socialization. *ACM HotMobile* (2018).

[71] Liang Zheng, Yi Yang, and Qi Tian. 2018. SIFT meets CNN: A decade survey of instance retrieval. *IEEE transactions on pattern analysis and machine intelligence* 40, 5 (2018), 1224–1244.

[72] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. 2017. Deep feature flow for video recognition. In *CVPR*, Vol. 1. 3.

[73] Zoran Zivkovic. 2004. Improved Adaptive Gaussian Mixture Model for Background Subtraction. In *International Conference Pattern Recognition (ICPR)*. IEEE, 28–31. https://doi.org/10.1109/ICPR.2004.479

[74] Zoran Zivkovic and Ferdinand van der Heijden. 2006. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters* 27, 7 (2006), 773 – 780. https://doi.org/10.1016/j.patrec.2005.11.005