DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models using Wearable Commodity Hardware

Akhil Mathur[‡], Nicholas D. Lane[‡†], Sourav Bhattacharya[‡]
Aidan Boran[‡], Claudio Forlivesi[‡], Fahim Kawsar[‡]
[‡]Nokia Bell Labs, [†]University College London

ABSTRACT

Wearable devices with in-built cameras present interesting opportunities for users to capture various aspects of their daily life and are potentially also useful in supporting users with low vision in their everyday tasks. However, state-of-the-art image wearables available in the market are limited to capturing images periodically and do not provide any real-time analysis of the data that might be useful for the wearers.

In this paper, we present DeepEye - a match-box sized wearable camera that is capable of running multiple cloud-scale deep learning models locally on the device, thereby enabling rich analysis of the captured images in near real-time without offloading them to the cloud. DeepEye is powered by a commodity wearable processor (Snapdragon 410) which ensures its wearable form factor. The software architecture for DeepEye addresses a key limitation with executing multiple deep learning models on constrained hardware, that is their limited runtime memory. We propose a novel inference software pipeline that targets the local execution of multiple deep vision models (specifically, CNNs) by interleaving the execution of computation-heavy convolutional layers with the loading of memory-heavy fully-connected layers.

Beyond this core idea, the execution framework incorporates: a memory caching scheme and a selective use of model compression techniques that further minimizes memory bottlenecks. Through a series of experiments, we show that our execution framework outperforms the baseline approaches significantly in terms of inference latency, memory requirements and energy consumption.

Keywords

Wearables; deep learning; embedded devices; computer vision; local execution

1. INTRODUCTION

Mobile vision systems have been built for decades, with recent example systems from research including SenseCam [1], Gabriel [2] and ThirdEye [3]. All of these systems capture images from devices worn by the user which are then processed by vision algorithms for extracting high-level information in service of a range of applications; for example, cognitive assistance, physical analytics, mobile health, memory augmentation and lifelogging. Now even commercial systems of this type are for sale with the most popular examples being Google Glass [4] and the Narrative Clip [5]. Fortuitously, as wearable cameras start to become mainstream the ability of computational models to understand the images they collect has also undergone a revolution in its abilities. Powered by rapid innovation in the area of deep learning for some image tasks (recognizing a face [6], locating an image geographically [7], or deciding if an image is memorable or not [8]) these models are even equalling human performance.

However, currently there exists a huge gap between this growing computational ability to understand images and the ability of wearable resource constrained devices to execute them. While this is an active area of research [9, 10, 11, 12, 13, 14], still today virtually no *wearable* devices in either research or sold commercial locally process the images they collect with deep learning models; and are limited, at best to processing them in the cloud (e.g., systems like [15]) which exposes users to privacy risks as highly sensitive personal images are handled by a third party. In fact, many devices like the Narrative Clip simply capture the image, transfer them to the cloud for later analysis – and perform no local computation on the image at all. Alternatively, research prototypes like Gabriel [2] that is based on Google Glass do perform some mixture of local and cloud-based processing of images using shallow image modeling algorithms (that are not as accurate as deep learning models); but, this processing comes at a cost with Gabriel having only 1 or 2 hours of battery life (reported in [2]).

The aim of this paper is to explore the future of wearable vision systems, where devices can locally process images with bleeding edge deep learning algorithms towards realizing a wide range of mobile vision based applications. In this work, we aim to not investigate what will it take for a *single* deep vision model to run efficiently – instead we attempt to run *multiple* concurrent deep learning algorithms using a wearable form-factor prototype. *We believe this is a critical system design issue that will enable important ubiquitous computing and mobile computing applications, many of which are already developed and studied [1, 16, 17, 18, 19, 20, 21], but remain impractical for large-scale deployment because suitable wearable devices are not yet available.*

In this work we present the design, implementation and evaluation of *DeepEye*; a one-of-a-kind wearable that is capable of executing 5 state-of-the-art deep vision models entirely on the device on images that are periodically captured (with a default sampling rate of 30 seconds). DeepEye is unlike any current wearable vision device in its capability to process images without cloud support, an ability that affords critical privacy assurances to users who

MobiSys'17, June 19-23, 2017, Niagara Falls, NY, USA

ISBN 978-1-4503-4928-4/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3081333.3081359

can trust their images never have to leave the device. In its current form the prototype device weighs 60gm and it has dimensions of 44 X 68 X 12 mm (illustrations are available in Figures 1 and 8). Although it has general support for all forms of Convolutional (CNNs) and Deep Neural Network models (DNNs) [22] (the most popular two forms of deep learning) – we evaluate it within this paper while supporting deep models that perform the following image tasks: face recognition, counting salient images in a scence, visual scene recognition, object detection, age and gender assessment of faces and more unusually the prediction of the memorability of an image [8]. Using these models it would be possible to use DeepEye to realize a wide range of vision applications; for example, a life-logging application that smartly filters images based on their expected memorability. Importantly, under this workload DeepEye still manages to maintain a battery life of nearly 17 hours with it taking only 10.10 seconds (depending on the model set used) to compute all deep models.

Two core enablers allow DeepEye to provide these levels of efficient mobile image processing. The first is the hardware design that combines a Qualcomm Snapdragon 410 processor (the same found in many smartwatches) with a custom integrated carrier board of our own design consisting of a 5 megapixel camera sensor and a 2400 mAh LiPo battery. The second enabler is an inference pipeline optimized specifically to cope with the needs of multiple models. Multiple deep model inference execution presents specific bottlenecks separate to those of single models; for instance, it becomes impossible to always keep all layers for each model in memory. Our inference pipeline optimizes *at the level of individual deep learning layers* and builds on a fundamental inter-model optimization insight: namely, that CNNs are comprised of a combination of computation-heavy convolutional layers and memory-heavy fully-connected layers. While convolutional layers tax the memory resources lightly, they are computationally demanding; in contrast, fully-connected layers have the exact opposite resource demands. Due to these orthogonal resource demands of memory and compute, it is possible to schedule and batch layers together from multiple models that better maximizes the resource of constrained devices and avoids bottlenecks that prevents multiple deep models from being executed. This core idea is complemented with a layer caching scheme and a selective use of SVD-based layer compression that further minimizes memory bottlenecks.

Finally – to complement our system and resource efficiency focus – we investigate the accuracy of the deep models used within DeepEye by testing them against images collected by actual wearable cameras (specifically Narrative Clips [5]). This is important because these models were not trained with data captured by wearable cameras and this will negatively impact their performance. Typically their accuracy numbers are assessed with data that is not representative of wearable devices. To address this, more than 18,000 images are provided from the authors of [23] which we have further labeled with image entities (≈4000) using Amazon Mechanical Turk (detailed in § 5.6). Overall, our findings indicate that deep models' accuracy does (as expected) decline than typical reported values primarily due to the nature of egocentric images (e.g., low lighting conditions, object occlusion). As such, there is a clear need to fine-tune these deep models for objects and scenarios commonly found in egocentric images.

The scientific contributions of this work include:

- We design the proof-of-concept DeepEye wearable. This is a first-of-its-kind device that is capable of executing multiple deep learning based vision algorithms purely locally. Comparable wearables from research, and those available commercially, at best may be able to run just a single example deep model – but none are designed to, or capable of, executing multiple models.
- We develop an inference pipeline for wearables designed to meet the challenges faced when multiple models are being executed (large memory footprint, shear computational overhead). This pipeline primarily relies on increasing processor utilization by scheduling layers with orthogonal bottlenecks.
- We build prototype hardware to demonstrate the feasibility of our design. This prototype relies on a commodity SoC (containing both CPU and GPU) and is capable of capturing/processing images with 5 deep models within reasonable resource bounds.
- We perform an extensive evaluation of the performance of DeepEye including its: inference pipeline, hardware, and even the accuracy of these models using images typical of wearable devices. We show that relative to alternatives, like serially performing inference using all models, the DeepEye offer more than 1.7x gains in execution latency.

Device	Weight (gm)	Lifetime (hours)	Cloud?	Inference?
DeepEye	60	16	No	Yes
Gabriel [2]	42	1	Yes	Yes
SenseCam [1]	93.5	24	No	No
Autographer [24]	59	24	No	No
Narrative Clip [5]	19	30	No	No

Table 1: Comparison of DeepEye with recent and/or popular wearables from the literature and available commercially

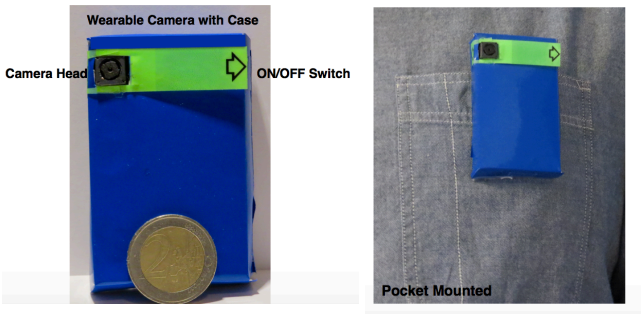


Figure 1: DeepEye Prototype. Device is comfortable being placed in the front pocket of the user's shirt.

2. DeepEye OVERVIEW

In this section we begin by systematically describing DeepEye and summarizing the capabilities of this wearable. DeepEye extends current capabilities of wearables by allowing simultaneous execution of multiple large-scale deep vision models locally in an efficient manner. The main use-cases of running multiple vision models include: (i) *logging of everyday* life, while capturing various aspects of the environment, (ii) providing navigation and social support for partially blind people in real-world situations, and (iii) enabling privacy-aware inferencing on personal devices by running all inferences on locally, completely avoiding cloud-based services. Moreover, as new sensors are becoming increasingly available on mobile platforms, application developers are adding new features that require a number of different inference tasks to be executed simultaneously.

2.1 Existing Camera Wearables

In Table 1, we summarize various camera-based wearable devices proposed in the literature as well as those commercially available

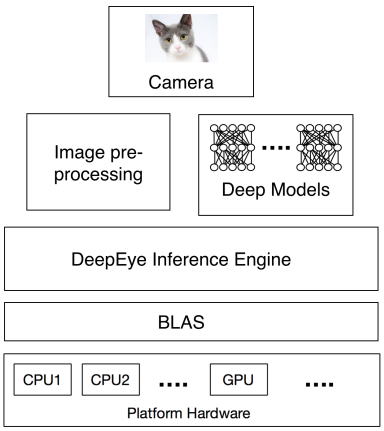


Figure 2: Software architecture of DeepEye

in the market, along with their form factors and capabilities. A common observation is that most camera-capture wearables such as Narrative Clip or Autographer only capture and stores images without providing any kind of context inferencing support. Further, devices such as the Gabriel [2] which provide inference on the collected images do so by offloading computations to a cloud backend. Contrary to existing wearables, DeepEye makes a significant leap forward by addressing technical challenges in simultaneously executing a number of very large vision models locally, which has not been attempted previously. Figure 1 illustrates a fully functional prototype of DeepEye.

2.2 Design Considerations

The primary design goal for DeepEye is to support the execution of multiple deep vision models locally on the device, which can enable a new class of applications for camera-based wearables. In this vein, the following three key design considerations have shaped the design of DeepEye:

- **Multiple Model Execution:** Study of how a system will behave, and what needs to be optimized when a number of models need to be locally executed periodically on the device. This means no single model can completely reside in memory constantly. This is also similar to the situation where a single large deep model is too big to reside completely at memory, this can become an increasingly critical issue due to the direction of deep models to be 100s and even 1000s layers deep – which is much larger than the 10 to 20 layers more often seen today.
- **Wearable Continous Vision Device:** The device should have a battery life able to last the typical waking hours of a day, within a form-factor that is comfortable for the user. Ideally a form-factor not significantly larger than existing devices today (such as the Narrative Clip) but still enabling a transformative increase in the amount of vision based deep models applicable to collected images.
- **Minimal Accuracy Loss Optimizations:** The device should minimize the accuracy losses associated with common optimization techniques which trade-off accuracy for resource reduction. As such, the core techniques we target in our design seek to increase the utilization of the device SoC and the efficiencies available in multiple model situations.

2.3 Architecture and Dataflow

We briefly describe the operation of DeepEye to provide context for how data is processed, and thus how the optimizations discussed in the following section are applied. Figure 2 depicts the overall architecture of our system, and its main components are as follows:

- **Layer Store Initialization:** Before DeepEye ever runs it must be initialized based on the collection of deep models to be executed. Each layer of the model is separated, and dependencies from the model architecture noted. DeepEye treats layers as the unit that are processed and once all layers of a model are processed then the output of the entire model is determined. At this phase logical dependencies between models can also be configured. For example, some models may only operate on face, therefore they should only be activated when a face is detected (by a face recognition model). Note: For the purposes of experiments all models execute regardless of such dependencies.
- **Sampling and Pre-processing:** At a configurable sampling rate images are taken and held temporarily for each deep model to be applied. At this phase the various types of pre-processing required by each model are applied so that their input layer can be initialized when it runs.
- **Inference Engine:** As described in the next section, this engine manages when different CNN layers are processed and how layers are cached to reduce overhead such as the paging in and out of layers. Individual layers are executed by DeepEye with the aim to reduce the resources consumed by all models collectively. The execution time of a total collection of models defines an upper bound on the camera sampling frequency; it is prohibited for the camera to sample photos at a rate faster than the system can process them locally. DeepEye can offer image inferences in the form of local API calls that applications can then be built.
- **BLAS:** In order to optimize the numeric operations on the hardware, DeepEye uses the optimized BLAS library for numeric computations. The inference engine passes the layers to be executed on the hardware to BLAS, which in turn distributes their computations across the multiple processors available on the device.

3. MULTIPLE MODEL DEEP INFERENCE PIPELINE

In this section we describe the inference pipeline of DeepEye. We begin by providing a primer on a Convolutional Neural Network (CNN) which is a popular deep learning model architecture for computer vision models. Next, we describe the major bottlenecks in running one or multiple CNN models on resource-constrained devices. To address these bottlenecks, we propose a new execution pipeline which includes a number of optimization strategies and a novel scheduling technique.

3.1 CNNs on Embedded Devices

Here we focus on the challenges associated with running CNN models on resource-constrained embedded devices. We begin by providing a brief overview of how a CNN model generates inference from the raw data, and then explain the challenges in executing the various layers of a CNN on embedded devices such as DeepEye.

Convolutional Neural Network Primer. CNNs are an alternative to DNNs (Deep Neural Networks) that still share many architectural similarities. As shown in Figure 3, a CNN is composed of one or more: convolutional layers, pooling or sub-sampling layers, and fully connected layers (with this final type being equivalent to those used in DNNs). The aim of these layers is to extract simple representations at high resolution from the input data, and then

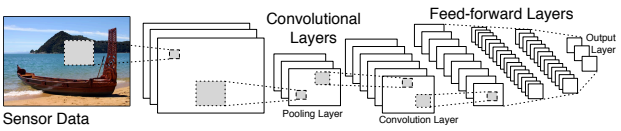


Figure 3: A CNN mixes convolutional and feed-forward layers.

converting these into more complex representations, but at much coarser resolutions within subsequent layers. This is achieved by first applying convolutional filters (with small kernel width) to capture local data properties. Next follow max or min pooling layers causing representations to be invariant to translations, this also acts as a form of dimensionality reduction. Finally, fully connected layers (i.e., a DNN) are applied to complete classification. Inference under a CNN operates only on a single segment of data (i.e., an image) at a time. Data is provided to convolutional layers at the head of the architecture. This can be considered a form of feature extraction before the fully connected layers are engaged. Inference then proceeds exactly as previously described for DNNs until ultimately a classification is reached.

CNN bottlenecks on resource-constrained devices. As has been reported in prior literature [9], the memory overhead in executing a deep model is dominated by the loading of weight parameters of fully-connected (fc) layers. In comparison, the convolutional kernels are small is size and take significantly less time to load. On the other hand, the computational overhead in deep model execution is dominated by the convolution operations, which are an order of magnitude higher than the matrix multiplication operations in the fc layers. In the context of wearable and embedded devices, the slow disc read and memory bus speed could be a major bottleneck is loading FC layers into the memory, and increase the latency of model execution. We study this bottleneck through a experiment on eight popular vision-based CNN models, where we run the models on an embedded platform (Qualcomm Snapdragon 410c) and measure the load and execution times of all the individual layers. In Figure 4 we summarize the results from this experiment and show the breakdown of average[1] time taken to load the fully-connected layers and execute the convolutional layers. The experiment reveals that on this embedded platform, the loading of fully-connected layers is the most expensive operation, even more than the loading and execution of convolutional layers by a factor of 1.3 (ObjectNet) to 2.15 (MemNet). As such, if we can explore opportunities to reduce the loading time of FC-layers, it will speed up the entire inference pipeline. In §3.2, we propose employing layer caching and layer compression techniques on the FC-layers to achieve latency gains in execution of deep models on constrained devices.

Bottlenecks in running multiple CNNs on embedded devices. As the demand for simultaneously running a number of deep models increases, we quickly run into a situation when the available memory on an embedded platform becomes insufficient to hold all the deep models. Under this situation, a common strategy is to run the deep models sequentially, i.e., loading one model at a time, run inferencing and clearing model memory before moving on to the next model. However this repeated paging-in and paging-out of large models adds significant overhead to the inference process, especially in the case of continuous inference systems such as DeepEye. Moreover, there can be cases where a deep model is so large that it cannot fit into the available memory on an embedded platform – in these scenarios, a common approach is to break the deep

[1] All models have been executed 100 times and we report the average here.

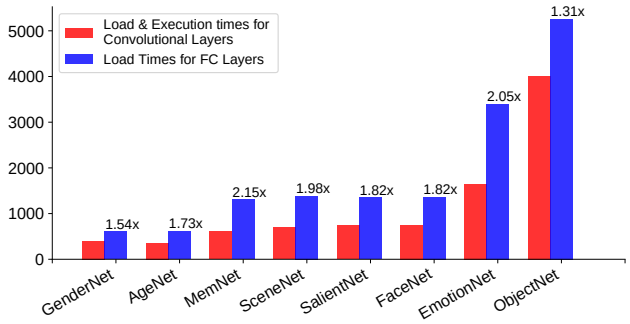


Figure 4: Understanding the bottleneck in deep vision model execution on embedded platforms. Loading model parameters is the single most time consuming task, which was seen across eight different models popular within the computer vision community.

model into its constituent layers, which are then paged-in to memory and executed in series. Even then, the bottleneck associated with loading of model parameters into memory still exists, and in fact it is amplified due to the presence of multiple deep models. In §3.2, we present a new technique of scheduling the execution of deep model layers which provides latency gains over the series execution – our approach exploits a key property of CNN models which is that the execution of layers follow a strict order (i.e. fc layers are executed on the output of the convolution layers), and aims to interleave the execution of convolution layers with loading of FC layers.

3.2 DeepEye Inference Engine

In this section, we present an overview of the inference pipeline used by DeepEye for executing multiple CNN models on resource-constrained devices. The key innovation in the pipeline is a novel approach of interleaving the loading of fully-connected layers with the execution of convolutional layers. We also discuss how our inference pipeline incorporates layer caching, model compression, and image hashing (detailed in § 3.3) to provide additional latency gains over the baseline approaches. Note that the pipeline is independent of the underlying processor, and can be applied to any available processor (e.g., CPU, DSP) on the system.

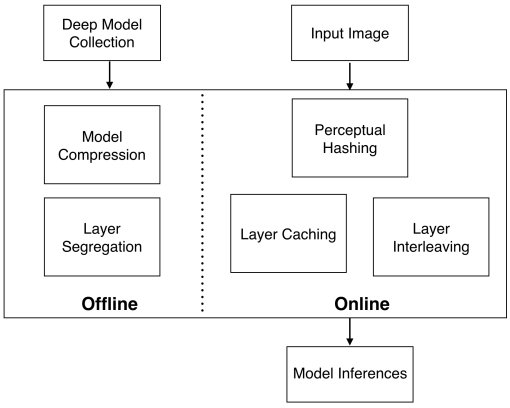


Figure 5: Overview of the DeepEye Inference Engine

Figure 5 illustrates the various components of the deep inference engine. The inference engine is initialized offline by segregating all model layers into computation- and memory-heavy layers, i.e. convolutional and fully-connected layers respectively. Next, the engine allows various kinds of model compression techniques to

be applied on the pool of layers. Currently, DeepEye supports the SVD-based layer compression technique proposed in the DeepX toolkit [25] for model compression, however more techniques could be added in future. In the online phase while computing the inference, the engine reads an input image from the camera, calculates a perceptual hash vector [26] (detailed in the subsequent sections) for the image, and compares it with the hash values of the last five images. If the hash similarity is above a certain threshold (i.e., the images are similar), the inference pipeline is terminated and the inference output corresponding to the nearest hash vector is returned. If the image hash does not match the previous five images, the inference pipelines continues by applying various pre-processing steps on the image to generate the necessary input expected by the individual models in the pipeline. Once the preprocessing of the image is completed, the results are immediately fed to the input layer (convolutional layers) of all the models. Thereafter, the caching and interleaved execution of convolution and FC layers take place (detailed in § 3.3), which finally results in an inference output for each model. The inference results are then written to the disc or can be passed onto any user-facing apps through API calls.

3.3 Optimization Techniques

In the following subsections, we discuss each of the optimization techniques employed by the DeepEye engine.

Runtime Interleaving: The core runtime optimization in DeepEye's execution pipeline is based on the idea of processing different layers from a pool of deep vision models in parallel to reduce the execution latency. As discussed above, the loading of FC layers and the execution of convolution layers are the two most expensive operations in running a deep model on embedded devices. Interestingly, due to the feed-forward inferencing followed in deep models, the memory-heavy FC layers are only executed after the input is fully processed by the convolution layers – as such, the loading of FC layers can potentially be interleaved with the execution of convolution layers. Naturally, the case of multiple model inferences allows for greater opportunities to parallelize the aforementioned execution and loading processes across multiple models.

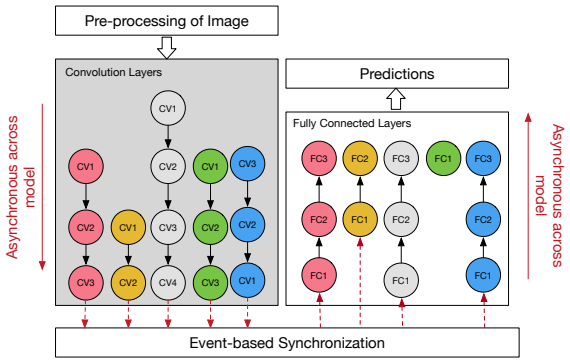


Figure 6: Runtime interleaving of convolutional and FC layers

Figure 6 depicts the overview of the runtime interleaving for an example case of five deep models. As indicated before, we first segregate all model layers into two pools, namely convolution layers and fully-connected layers. We next spawn two threads, namely *convolution-execution* thread and *data-loading* thread. In the convolution-execution thread, the convolution filter parameters of all models are loaded into the memory and the convolution operations begin on the pre-processed input data for each model. We take advantage of the underlying BLAS numeric library used in our implementation (detailed in § 4.2) to perform efficient multi-threaded numerical operations (e.g., convolutions) – as such we do not spawn separate convolution-execution threads for each model. Instead we adopt a FIFO queue based execution strategy for the convolutional layers across models. For example, as shown in Figure 6, three layers of convolution operations for red model are carried out by this thread, which are followed by two layers of yellow and three layers of the grey model. The *data-loading* thread which is spawned in parallel with the convolution thread is responsible for loading the FC layer parameters for all models into the memory, again in a pipelined manner (i.e., one model after the other).

The objective of the *convolution-execution* thread is to perform all convolutions on the input image, and pass the results of the final convolution layer of each model to the *data-loading* thread. When the *data-loading* thread finishes loading the FC layer parameters for a model, it can use the pre-computed convolution outputs from the *convolution-execution* and proceed to obtain the final classification results.

The latency gain G from runtime interleaving can be computed as:

$$G = \frac{t_{conv} + t_{fc\_load}}{max\{t_{conv}, t_{fc\_load}\}} \tag{1}$$

where, t_{conv} represents the time taken to load the convolution filters and execute the convolution operations for all models, and t_{fc_load} represents the time taken to load all FC layer parameters into the memory. As such, the numerator in Equation 1 corresponds to the baseline case where convolution execution and loading of FC layers is done in series, while the denominator corresponds to the interleaved execution where the total time is the maximum of both threads' execution times. Note that the eventual execution of FC layers on the output generated from the convolutional layers in common in both series and interleaved approaches – as such, it is not accounted for while calculating the latency gain (G) due to interleaving.

The denominator of equation 1 can be simplified using the following equation which expresses the maximum of two numbers in terms of their sum and difference:

$$max\{a, b\} = \frac{1}{2}(a + b + |a - b|) \tag{2}$$

By substituting (2) in (1), we get:

$$\begin{aligned} G &= \frac{2}{1 + \frac{|t_{fc\_load} - t_{conv}|}{t_{fc\_load} + t_{conv}}} \\ &= \frac{2}{1 + \lambda} \end{aligned} \tag{3}$$

where $\lambda = \frac{|t_{fc\_load} - t_{conv}|}{t_{fc\_load} + t_{conv}}$ stands for the *interleaving factor* and captures the difference between t_{conv} and t_{fc_load}. As λ increases, the interleaving gain G decreases – and when $\lambda = 0$ (i.e., time taken to load the FC layers is the same as time taken to run the convolutions), the gain G is maximized to 2x. In § 5, we experiment with different values of λ to showcase interleaving gains in multiple scenarios.

Caching of Memory-Heavy Layers: For continuous-sensing applications and devices such as DeepEye, caching of layer parameters into the memory could be potentially useful, as it saves significant time required to load thousands of parameters into memory for each inference. As loading the parameters for fully connected layers is one of the main time-consuming operations in the inference

process, the idea strategy would be the keep all parameters of these FC layers into memory. However, the limited runtime memory on the embedded devices becomes a bottleneck to load all the layer parameters, more so in the case of executing multiple deep models.

DeepEye adopts a greedy approach for caching the layers parameters of the FC layers to maximize memory utilization. It monitors the available runtime memory and caches the k largest FC layers from the model pool that can fit into the memory. In case the available memory cannot fit all k layers, it caches $k-1$ largest layers, and again greedily repeat the process to cache the second-largest layers into the remaining memory. As layer caching effectively reduces the total load time for the FC layers (t_{fc_load}), it is likely to reduce the value of λ and increase the latency gains by interleaving.

Model Compression: In addition to layer caching, another potential approach to reduce the load times of the memory-heavy FC layers is to employ model compression techniques. The goal of model compression techniques is to reduce the size and execution latency for a single deep model, at the expense of some accuracy loss. In our work, we use the SVD-based layer factorization approach proposed in the DeepX toolkit [25, 27] to compress the FC layers of all deep models. The SVD-factorization technique takes the weight matrix containing the matrix multiplication parameters for two adjacent FC layers, and factorizes it into two matrices which together contain lesser parameters than the original matrix. As a result of this factorization, the total number of parameters in the FC layers are reduced, which in turn lowers the load time of the FC layers (t_{fc_load}). This also reduces the value of λ (as the difference between t_{fc_load} and t_{conv} reduces), and hence increases the latency gains (G) from interleaving. However, it is interesting to note that if too much compression (C) is applied to the FC layers such that $t_{conv} > C * t_{fc\_load}$ i.e., if the convolution execution time starts dominating the FC loading operation, then the gains due to compression will no longer be observable.

Pre-empt using Hashing: In continuous image capture systems such as DeepEye, it is likely that consecutively captured images are similar in their content – this situation often arises in everyday life logging, for e.g., when the wearer is stationary or the capture device is kept on a table. If the system can identify such similar images, it need not run the entire inference pipeline on the image, and a significant amount of computations and energy could be saved over time. Although, stationary conditions can be adequately detected by inertial sensors like accelerometer, in this work we rely on an image-based hashing technique to detect image similarity. More specifically, we apply a *perceptual hashing* algorithm [26] on the input images and store hash values of the last five images along with their predictions. Perceptual hashing functions are widely used to establish the *perceptual similarity* of multimedia content – this is done by extracting a series of features from the content (i.e., the raw image) and applying a hash function on those features. DeepEye uses the Radial Variance Based Hash function as implemented in [26]. When a new image is captured, we first compute the hash of the image and then calculate its *hamming* distance to all five previously stored hashes. If the hamming distance is found to be smaller than a predefined threshold, we then pre-empt execution of all deep models and return the classes stored for the image with smallest hamming distance. In our implementation we use a distance threshold of 26, which is suggested by the developers of the hashing algorithm.

4. EMBEDDED IMPLEMENTATION

In this section we describe the hardware implementation of DeepEye with a quad-core Qualcomm Snapdragon 410 processor. We also describe our current software implementation designed to run multiple deep models on DeepEye using the Torch framework.

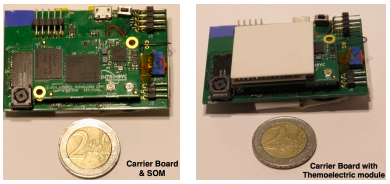


Figure 7: DeepEye Hardware Prototype

4.1 Hardware Prototype

DeepEye is powered by a quad-core Qualcomm Snapdragon 410 processor on a custom integrated carrier board, and includes a 2400 mAh LiPo battery. We intend to open-source the design of DeepEye hardware soon to help other researchers build similar wearable devices.

Snapdragon 410 Series APQ8016. The DeepEye wearable camera consists of a System On Chip (SOC) mounted on a carrier board as shown in Figure 8. The SOC has a quad core ARM processor (Snapdragon 410 Series APQ8016 from Qualcomm), a Qualcomm Adreno 306 GPU, 8GB of flash storage and 700MB of usable RAM. The carrier board allows the SOC to be plugged in and provides ports for a Camera Serial Interface (CSI) camera, a UART for programming and debug, i2C and SPI GPIOs for connecting sensors.

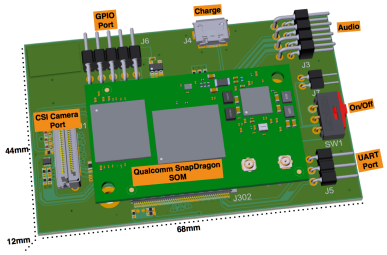


Figure 8: DeepEye Qualcomm SoC and Custom Integrated Carrier Board

Custom Integrated Carrier Board. The carrier board is powered by a 3.7v LiPo battery. The SoC provides power management and recharging circuitry while the carrier board provides a micro usb connector for charging. The DeepEye wearable runs a custom build of Linaro Debian linux based on the 4.3 Linux kernel. The custom build adds driver support of high-speed CSI camera and removes unused devices (HDMI bridge, DSI panel, USB hub) by using a custom Linux kernel and device tree. The camera module uses a 5 Megapixels Omnivision OV5640 camera sensor.

The DeepEye wearable has a small form factor (44 X 68 X 12 mm), and includes a 2400mAh LiPo battery. The device weights 24gm without the battery, and 60 gm with the battery attached to it, thus making it practical for everyday use.

4.2 Software Implementation

Our current software implementation for DeepEye uses an optimized version of the Torch deep learning framework for doing inference on the embedded device. The motivations for using Torch were the availability of state-of-the-art vision models for this framework, and its excellent portability to embedded platforms. By removing non-essential components from the Torch framework, we were able to keep these benefits while also reducing the overhead of the framework relative to a custom C++ runtime. We now discuss the major components of our implementation:

BLAS Runtime. While executing deep model inferencing, the main computational load remains in performing matrix multiplications [28, 25, 27]. In our current implementation of DeepEye we use the highly optimized BLAS library [29] for running the convolutions and the matrix multiplications needed in the fully-connected layers. In the current implementation we use Torch as our deep learning inferencing engine, which supports multi-threading for deep model execution. The Torch framework loads the model layers into the memory and send them to the BLAS library for computation. Thereafter, BLAS distributes the actual computational tasks across the CPU or GPU cores as specified by the model. Note that DeepEye is agnostic to the underlying deep learning framework and can be easily integrated with other engines supporting multi-threaded execution, e.g., TensorFlow and Theano.

Control Subsystem. We implement the data-flow runtime as described in §2 in the following manner. DeepEye camera captures an image every 30 seconds, and saves it to the disk. This image is then loaded in the Torch framework and a number of pre-processing operations are done on it. Although these operations are specific to each deep model, they broadly include cropping, resizing and mean normalization of the input image.

Next, the deep vision models are read from the storage and loaded into the shared memory in a layer-wise manner by Torch, that is each layer is paged-in the memory, where it operates on its input data and generates the input for the next layer. Thereafter, the layer is paged-out of memory and the next layer is paged-in. As described in §3, these load/execute are done in series in the baseline case and by applying interleaving (and other optimizations) in our proposed approach.

DeepEye is capable of using both the CPU and GPU on Snapdragon 410c for running convolution operations. The Snapdragon on-board GPU however lacks OpenCL support, therefore we implemented an Open-GL based Convolution Engine (described in next subsection) which can run convolution operations using OpenGL shaders. When the system has to compute a convolution (which is the dominant operation in the initial layers of a CNN), it can execute the operation on the quad-core CPU (using Torch) or on the GPU (using the Open-GL engine).

Open-GL Convolution Acceleration. OpenGL is a 3-D drawing specification which allows applications to accelerate graphics using an accelerated rendering pipeline implemented in the GPU. Recently a subset of OpenGL, named OpenGL ES, has made it possible for other computational algorithms to execute on the GPU, if they could map their data into graphical objects, such as bitmaps, textures and vertex buffers. In the case of deep learning models, our key idea is to represent the input to a convolution layer as a bitmap object, then apply an OpenGL Shader to it to convert it to another bitmap, which is then fed to the subsequent layers.

5. EVALUATION

In this section, we first benchmark the performance of various optimization techniques described in §3 viz. layer interleaving, layer caching, and model compression against the baseline approach of executing deep models in a layer-wise fashion. Next, we present two case studies inspired by real-world applications of DeepEye and highlight latency and energy gains achieved by our proposed inference pipeline. Note that we take a CPU-centric approach in our experiments – i.e., all experiments were conducted on the on-board quad-core CPU of DeepEye. Later, we also discuss using the on-board GPU for running the inference pipeline and highlight why it does not add to any latency gains. Our main findings can be summarized as follows:

- DeepEye can last for more than 33 hours on a single battery charge, assuming an image is captured and analyzed by the deep learning models every minute.
- Our proposed execution approach of layer interleaving and layer caching results in nearly 2x gains in runtime over the baseline method of series execution.
- The interleaved execution technique could work in tandem with model compression, and serve as a way to offset the accuracy losses due to compression while providing comparable latency gains.

5.1 Methodology

We used the Qualcomm Dragonboard 410c, which is the development board for the Snapdragon 410 processor to perform the experiments for evaluating the performance of DeepEye and its optimization techniques. For energy measurements, we instrumented the power lead of the Dragonboard 410c to connect an Ammeter to it. We also disabled those peripherals (e.g. HDMI port, SD Card, WiFi chip) on the Dragonboard which are not present on DeepEye to get accurate energy measurements. As representative workload for our experiments, we used eight state-of-the-art pre-trained deep learning models trained for a variety of computer vision tasks. In the next section, we present more details about the deep models used for evaluation.

Deep Models: In total, we used eight CNN models of varying sizes and computational complexities that are representative of common vision tasks. All these models are pre-trained and available as open-source models from Torch or Caffe framework websites. A summary of these models is also presented in Table 2.

FaceNet [30]: Face detection is a popular task for any image capturing system, and is a necessary step for algorithms which aim to detect smile or emotions in an image. Here, we use a face detection CNN model that can detect faces in a wide range of orientations.

AgeNet and GenderNet [31]: AgeNet and GenderNet models aim to predict the age (8 categories) and gender (2 categories) of a person from an input image. The models have been trained using around 20K images.

EmotionNet [32]: This model aims to recognize the emotion on a person's face by analyzing the input image. The input to this model is a cropped RGB image of the face, and it outputs confidence scores for 8 different emotions: angry, disgust, happy, sad, surprise, fear and neutral. The authors of this model also offer additional emotion recognition models, all of which when used together offer state-of-the-art performance – however, for the sake of experimentation - we only use the model that operates on RGB values.

SalientNet [33]: This model aims to compute the saliency of an input image, by predicting the existence and the number of salient objects in a given scene. Models such as SalientNet could be particularly useful to filter out non-salient images from the lifelog outputs.

ObjectNet [34]: Detecting various objects present in an image is of utmost importance to both our representative use-cases. For this, we consider a popular CNN model named VGG which supports more than 1000 object classes (e.g., dog, car).

MemNet [35]: The aim of this model is to determine how memorable an image is (with actual output being a "memorability score"). Experiments indicate MemNet has the ability to select images that

Scenario	Model	Size	Architecture
Lifelogging	SceneNet	240MB	c:8^{z}; p:3‡; fc:3*
Lifelogging	MemNet	233MB	c:8^{z}; p:3‡; fc:3*
Lifelogging	SalientNet	233MB	c:8^{z}; p:3‡; fc:3*
Vision Support	AgeNet	46MB	c:3^{z}; p:3‡; fc:3*
Vision Support	GenderNet	46MB	c:3^{z}; p:3‡; fc:3*
Vision Support	EmotionNet	419MB	c:5^{z}; p:3‡; fc:3*
Both	FaceNet	230MB	c:8^{z}; p:3‡; fc:3*
Both	ObjectNet	553MB	c:13^{z}; p:5‡; fc:3*

^{z}*convolution layers*; ‡*pooling layers*; **fully connected layers*

Table 2: Representative deep models and scenarios used for evaluating DeepEye

are memorable with similar success to those of human labelers. This model could also be potentially useful to assist lifeloggers in browsing through the thousands of images captured by DeepEye.

SceneNet [36]: A total of 476 scene types are recognized by this model. Example scenes include: bedrooms, kitchens, forests.

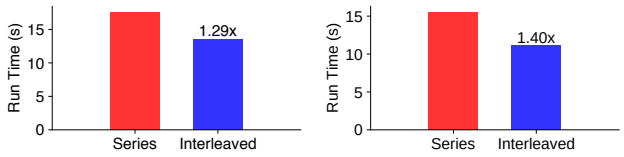
Scenarios. We logically group the deep models into two scenarios to illustrate practical use-cases of DeepEye, namely *i) lifelogging*, and *ii) vision support*. Lifelogging, i.e. capturing and archiving our everyday lives, has been a primary use-case for camera-based wearable devices such as the Microsoft SenseCam and Narrative Clip. For demonstrating a lifelogging scenario with DeepEye, we use models which can detect objects, places, faces in an image, along with inferring the saliency and memorability of the image. Another important application for camera wearables could be to support users with visual impairment, in that the wearable can analyze the captured images and provide non-visual (e.g. audio or haptic) feedback to the user. For demonstrating this scenario, we chose models that could detect the faces or object in an image, along with inferring the age, gender and emotion of people in the image. Both scenarios along with the models associated with them are summarized in Table 2.

5.2 Interleaving Analysis

Here we evaluate the impact of layer interleaving on the overall inference time of deep models. We first show the gains achieved in inference time by using layer interleaving in both application scenarios mentioned earlier. Then we empirically validate the sensitivity of interleaving gains as shown in Equation 3, by varying the value of the *interleaving factor* λ.

Setup. To evaluate the benefits of interleaving, we run the five deep models in each scenario in both series (baseline) and interleaved manner. No other optimization (e.g., caching, compression) is applied to the models. Next, in order to explore the sensitivity of interleaving as per Equation 3, we constructed model pipelines of various sizes in both scenarios. For example, in a scenario with 5 models, there are $^{5}C_{2}$ possible pipelines of 2 models in it, $^{5}C_{3}$ pipelines of 3 models and so on. In total, we generated 52 different pipelines for both scenarios, with each pipeline having a different length and combination of models. We executed each pipeline in the baseline Series condition and calculated the value of *interleaving factor* λ for it. Then we ran each pipeline in an interleaved way and calculate the inference time gain over the baseline condition. Below we present our findings for both the experiments.

Results. Figure 9 illustrates the reduction in inference time in the interleaved case for both scenarios. We clearly observe a reduction in inference time in both scenarios (1.29x in Lifelogging scenario and 1.4x in the Vision Support scenario), although the magnitude of reduction is different. To explore it further, in Figure 10, we plot



(a) Scenario: Lifelogging **(b)** Scenario: Vision Support

Figure 9: Latency comparison between series and interleaved execution. The values on the bars show the latency gains due to interleaved execution.

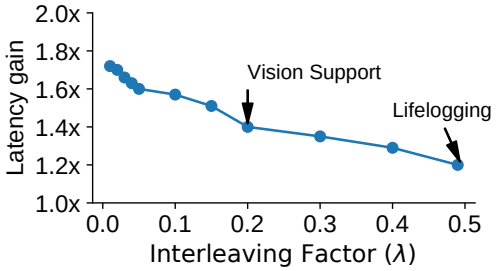


Figure 10: Latency gain under different values of λ

the latency gain (due to interleaving) against 10 different values of λ obtained from executing various combination of model pipelines. We can observe that as the value of λ increases, the latency gains due to interleaving decrease. As λ is proportional to the difference between execution time of convolution layers and load time of FC layers, this implies that if both these times are of the same order, the interleaving gains are maximized. For example, the Lifelogging scenario has a higher value of λ than the Vision Support scenario (due to the choice of models in them), and therefore exhibits lower latency gains.

To sum up, we found the interleaved execution provides a way to maximize the resource usage on the device, and provides latency gains without any loss in inference accuracy.

5.3 Interleaving Meets Model Compression

Our findings from interleaving analysis shows that although interleaving outperforms the baseline series execution, its latency gain has an upper bound of 2x when t_{fc_load} equals t_{conv} (Equation 3). However, this is difficult to achieve in practice as the loading times of FC layers are higher than the execution times of the convolution layers as shown in the model profiles in Figure 4. In this section, we investigate if model compression techniques proposed in the literature can reduce the load times of the FC layers, thereby increasing the latency gains in the interleaved execution.

Setup. For studying model compression, we choose four deep models from our model pool, namely SceneNet, MemNet, SalientNet, and FaceNet. On each model, we apply four levels of compression using the SVD-based weight factorization scheme proposed in prior literature such as the DeepX toolkit [25, 27, 37]. The SVD-based compression approach reduces the total parameters (and operations) in a fully-connected layer by factorizing it into two smaller layers, with some loss in overall inference accuracy. Our experiments apply four levels of compression (10%, 20%, 70%, 80%) to all FC-layers of the deep models, and we report the latency gains in each compression setting. The motivation to apply high compressions (70%, 80%) is to empirically show that the compression gains peak at a certain point, after which compression does not provide any additive gains to the interleaved execution.

Results. In Figure 11, we compare the performance of interleaved execution approach against series baseline and model compression

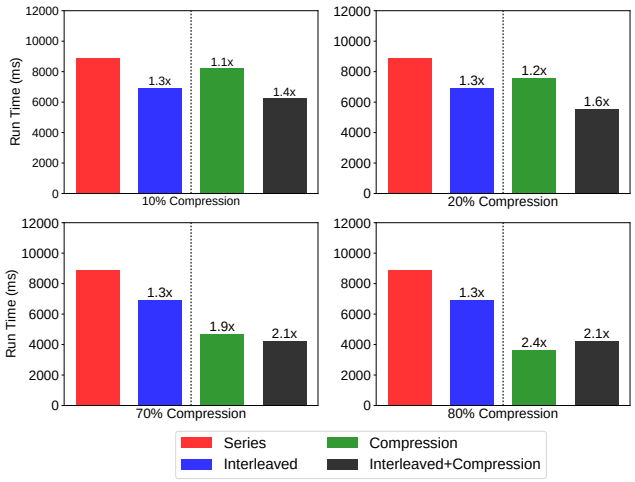


Figure 11: Performance of layer interleaving in combination with model compression approaches. The values on the bars show the latency gain in each case over the baseline Series execution approach.

for our selected 4-model pipeline. There are three key insights that emerge from Figure 11: firstly, as the compression ratio increases, the latency gain also increases for both compression-only approach and compression+interleaving, with the latter outperforming the other techniques. For example, in the 20% compression scenario, applying interleaving with compression provides a latency gain of 1.6x, as opposed to 1.2x provided by compression alone. Secondly, we observe that when the amount of compression applied to the model is too high (e.g., 80%), the interleaved execution no longer benefits from it, because the convolution execution starts dominating the overall runtime in the interleaved case. Finally, it is evident that interleaving can be used as a tool to offset against the accuracy loss caused by model compression. For example, we found that with 30% compression applied to the models, the pipeline can run 1.4x faster in the baseline case (not represented in the figure). However, the same 1.4x speedup can be achieved by applying 10% compression and interleaving the execution of layers as shown in Figure 11. As such, interleaving provides a way to get same latency gains with lesser compression and lesser accuracy losses.

5.4 Caching Analysis

In this section, we explore the benefits of caching memory-heavy layers on the overall runtime of the deep model pipeline. For continuous sensing wearables, caching of memory-heavy layer parameters into the memory could be potentially useful, as it saves significant time required to load thousands of parameters into memory for each inference.

Setup. For this experiment, four deep models from the Table 2 namely SceneNet, MemNet, SalientNet, and FaceNet are chosen as the workload. Although DeepEye has roughly ($\tilde{5}$70 MB) of available runtime memory, other low-end wearable devices often have much less runtime memory. As such, for a detailed evaluation of the caching approach, we simulate three more cases with runtime memory of 150MB, 300MB, and 400MB. In each case, we adopt a greedy approach to cache the n largest FC layers from the model pool into the memory. In case the available memory cannot fit all n layers, then we cache $n-1$ largest layers, and again greedily repeat the process to cache the second-largest layers into the remaining memory. In the following, we compare the latency gains by layer caching in series and interleaved execution approaches.

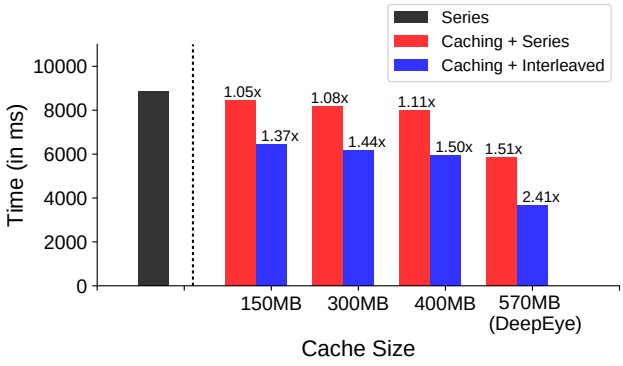


Figure 12: Performance of layer interleaving in combination with layer caching. The values on the bars show the latency gain in each case over the baseline Series execution approach.

Scenario	Latency per inference	Estimated Battery Life
Lifelogging	10.10s (1.74x gain)	33 hours
Vision Support	8.23x (1.88x gain)	43 hours

Table 3: Performance of DeepEye in the two application scenarios

Results. Figure 12 shows that as the runtime memory available on the device increases, we are able to cache more FC-layer parameters into the memory and hence achieve higher latency gains for both series and interleaved caching scenarios. For example, with 570MB available memory (as is the case on DeepEye), the caching-series execution approach gives a 1.51x latency gain whereas the caching-interleaved execution provides a 2.41x speedup over the baseline. Interestingly, it is worth noting that the interleaved-caching approach is able to provide a similar speedup (1.5x) with 400MB of runtime memory, whereas the caching-series approach would need an additional 170MB runtime memory to achieve the same speedup. As such, constrained devices with limited runtime memory can benefit by implementing the interleaved-caching mechanism for deep model execution.

The findings from the above experiments highlight the gains in inference latency on applying the various optimization techniques used by DeepEye. In the next section, we will present two case studies where we apply these optimizations on two real-world applications of continuous camera capture devices.

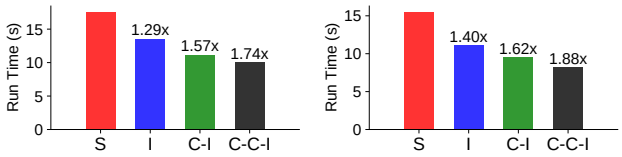
5.5 Case Studies

In this section, we evaluate the performance of DeepEye on the two scenarios described in previous section, namely *i) lifelogging*, and *ii) vision support*. Each scenario consist of five deep learning vision models as summarized in Table 2. We begin by presenting the overall performance when these scenarios are run on DeepEye, and then present detailed latency and energy results.

DeepEye Performance: We applied layer caching, compression, and interleaving to the pool of deep model layers and calculated the latency and energy consumed for performing one inference. Battery life is estimated from the per-inference energy consumption, assuming that an inference will be made every 1 minute. In Table 3, we show the per-inference latency and battery life of DeepEye when it executes each of the application scenarios. We observe the DeepEye provides at least 1.74x latency gain for both application scenarios, and the estimated battery life for the device is more than 33 hours.

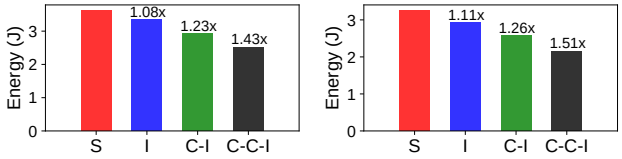
Execution time: Here we provide a detailed breakdown of the inference execution time for both the scenarios. For each scenario, we compare the baseline Series execution approach against the interleaved execution approach, with various optimizations (caching, compression) added to it. For scenario #1 (Lifelogging), we cached the largest fully connected layers of FaceNet, MemnNet and SalientNet into the memory, while for use-case #2 (Visual Impairment Support), we stored in memory the first fully-connected layers of VGG, GenderNet and AgeNet. As for model compression, we applied a 20% compression to each FC layer by using the SVD-based factorization approach discussed earlier.

In Figures 13a and 13b, we plot the total time taken to run the two model pipelines on the DeepEye CPU in four different execution conditions. The results are averaged over 10 iterations. We observe that the baseline approach of loading and running the layers in series without caching has the worst runtime performance – the time taken to run the entire inference pipeline being 17.57 seconds (in Scenario 1) and 15.51 seconds (in Scenario 2). On the contrary, layer interleaving in combination with caching and compression outperforms all other approaches – the runtime for the model pipeline in Scenario 1 and Scenario 2 are 10.10 seconds (~1.74x faster than baseline) and 8.23 seconds (~1.88x faster than baseline) respectively.



(a) Scenario: Lifelogging **(b)** Scenario: Vision Support

Figure 13: Latency comparison when executing the scenario pipelines on DeepEye. S = Series, I = Interleaved, C-I = Caching+Interleaved, C-C-I = Compression+Caching+Interleaved



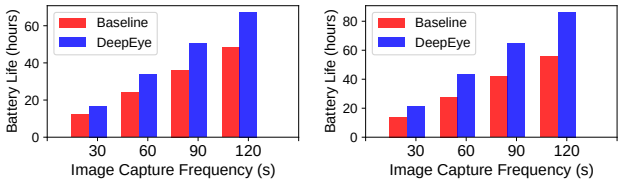
(a) Scenario: Lifelogging **(b)** Scenario: Vision Support

Figure 14: Energy comparison when executing the scenario pipelines on DeepEye. S = Series, I = Interleaved, C-I = Caching+Interleaved, C-C-I = Compression+Caching+Interleaved

Energy Performance. We measured the average current flow during each execution of a model pipeline, and used it to calculate the total energy consumption. In Figures 14a and 14b, we plot the energy consumed to run the two scenarios on DeepEye in various experiment conditions. Our findings show that by applying caching and compression in the interleaved execution case, DeepEye requires nearly 1.43x less energy than the baseline approach. Although the observed energy saving for one single execution may not be considered significant, these savings get compounded over time and can result in few additional hours of battery life as we will discuss next.

In Figure 15, we show the effect of image capture frequency on the expected battery life of DeepEye. These calculations were done by adding the energy consumption during idle time, energy cost of taking a picture, and the cost of one execution of the model pipeline. We observe that in the case of an image being captured and analyzed every 30 seconds, DeepEye's battery will last 16.85 hours for the Lifelogging scenario and 21.6 hours for the Vision Support scenario using our proposed optimization techniques. In a more reasonable case of an image being captured and analyzed every 1 minute, the battery lasts for 33 hours (9 hours more than the baseline) for Lifelogging and 43 hours (12 hours more than baseline) for the Vision Support scenario.

Hashing Performance. Now we present the findings of applying the *perceptual hashing* technique (discussed in §3) on a dataset of 18,735 egocentric images as described in §5.6. We chose a window-size of 5, that is we compute and store hash values of five recent images in the workload. For each new image, we compute its hash and calculate its hamming distance to all previously stored hashes. With a distance threshold of 26 (as suggested by the developers of the hashing algorithm), we found that for roughly 18% of all images in the dataset, we could pre-empt the execution of all deep models and return the classes stored for the image with smallest hamming distance. This approach eventually results in significant energy savings, and increases the battery life of the device.



(a) Scenario: Lifelogging **(b)** Scenario: Vision Support

Figure 15: Comparison of estimated battery life of DeepEye assuming different image capture frequencies. This does not include energy gains from perceptual hashing, which will increase the battery life of DeepEye.

GPU execution. Finally, we evaluated the runtime of our model pipelines by running them on both CPU and GPU. While executing the models on a GPU, we observed a 1.6x gain in execution of the convolution layers over CPU-only approach. However, as the CPU and GPU have a shared memory on the Snapdragon 410c, the loading time of fully connected layers into the memory remains the primary bottleneck. As such, even with faster convolution operations on the GPU, we do not gain much in terms of the overall inference time due to the bottleneck of loading fc layer parameters into the memory – moreover, using the GPU consumes additional energy which reduces the battery life of the wearable. Therefore, in the current version of DeepEye, we only employ the CPU for doing model computations.

5.6 Inference Accuracy

In this subsection we discuss the accuracy of the deep learning models used in our experiments on egocentric images. Note that the deep models used in our experiments were pre-trained on a wide variety of images, not specifically on egocentric images captured from a wearable. As such, we would like to understand how well these models perform on egocentric images, and whether they need extensive re-training in the future.

Dataset. For our evaluation, we used the egocentric image dataset from the authors of [23], which consists of a total of 18,735 images captured by 7 users wearing a Narrative Clip for 20 days. The Narrative Clip has the same camera resolution as DeepEye, therefore its images are representative of what will be captured by DeepEye. The dataset is rich in diversity, as the users were wearing the Narrative Clip in different contexts: while attending a conference, on

Figure 16: Examples of egocentric images captured from a Narrative Clip

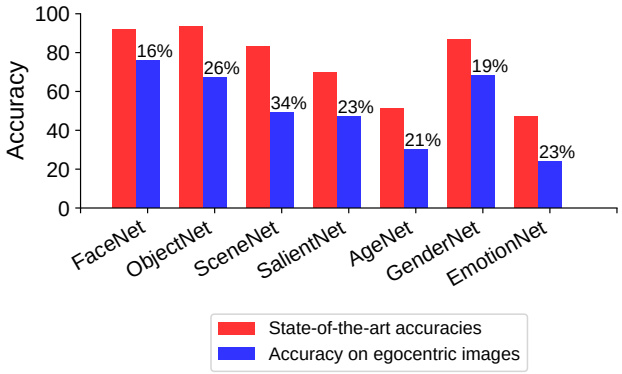


Figure 17: Accuracy of the pre-trained deep models on egocentric images. Numbers on the bars represent accuracy loss compared to the state-of-the-art accuracies, which happes because the models are not fine-tuned for egocentric images.

holiday, during the weekend, and during the week. A sample of the images in the dataset is shown in Figure 16.

Ground Truth Collection. The dataset from [23] is largely unlabeled for evaluating the models used in our experiment, with the exception of face labeling. For a subset of 4912 images in the dataset, the authors have labeled whether the image consists of a 'face' or not. As such, these images could be used to evaluate the accuracy of FaceNet in our experiments. To collect the ground truth for evaluation of other models, we ran two crowdsourcing tasks on Amazon Mechanical Turk.

In the first task, we asked workers to choose labels for the face images (extracted from the original dataset) viz. their age (8 age categories), gender (male/female) and emotion (7 categories). The label choices were the same as the output classes in AgeNet, GenderNet and EmotionNet. In the second task, we asked workers to label 4000 different images from the dataset viz. the number of salient objects in them (for evaluating SalientNet), objects seen in the image (for evaluating ObjectNet) and the places seen in the image (for evaluating SceneNet). While the original ObjectNet and SceneNet models consist of 1000 and 205 classes respectively, we only asked the workers to choose from object and scene classes that are expected to be captured in routine life (e.g. laptop, office, home, building, cup, fruit). We did not evaluate the accuracy for MemNet as it was difficult for crowd workers to label the memorability of an image captured by somebody else.

Results. Before presenting the results, it is worth pointing that none of the models used in our experiments were fine-tuned for egocentric images. Egocentric images typically have poor lighting conditions, multiple objects in them and they may even have the object or person of interest occluded by other objects – thereby making it challenging to get an accurate inference.

In Figure 17, we plot the accuracy of the models as observed on the egocentric image database. For ObjectNet and SceneNet, we present the top-5 accuracies, that is the ground-truth category is within the top-5 predicted categories. For other models, we present the top-1 accuracy. We observe that the face-based models, such as FaceNet, AgeNet, GenderNet, and EmotionNet have lower accuracies drops (numbers shown on top of the bars) as compared to the state-of-the-art accuracies. On the other hand, models which aim to analyze the scene (e.g., SceneNet or ObjectNet) reported much higher accuracies drops. Specifically for SceneNet, images captured indoors (e.g. at home in the evening) had a high number of misclassifications – we believe that the poor indoor lighting conditions made it challenging for the classifier to infer the location. Overall, these results suggest that fine-tuning the pre-trained models on egocentric images is required to achieve their reported accuracies.

6. DISCUSSION

We now discuss a key set of issues related to the design and evaluation of DeepEye.

Beyond Just Vision-based Deep Models. We believe the optimization techniques described will operate successfully even in models that are not image based. Our design of DeepEye and the optimizations we discover exploits characteristics of layers (such as fully-connected feed-forward layers in relation to other layer types) that should be suitable for other mobile platforms that target other uses of deep learning. For example, audio understanding and emotion recognition.

Relevance of Deep Learning Specific Accelerators. Existing accelerators such as [38, 39] are not able to execute multiple deep models as DeepEye is able to. While their performance for individual models we use (such as AlexNet and VGG based models) may execute more efficiently on this purpose-built hardware, how they will support multiple model situations remains unstudied.

Complementary to Many Existing Optimizations. Because model-centric techniques that use sparse-coding [27], SVD [40], node pruning [41] alter the layers of deep models themselves they are compatible with scheduling and caching approaches we develop here. As we show in the evaluation when the proposed methods and such model compression techniques are combined, the performance impact with respect to the accuracy trade-off can be impressive and better than the trade-offs possible with the original technique itself.

User Trials of DeepEye. Currently DeepEye has not yet been deployed. The focus of this work has been the development of hardware and software techniques that at this stage are evaluated through a trace-dataset. However, we will begin deployments of these devices for limited periods of time moving forward.

Improving Deep Model Accuracy. In this work we report some of the first observations of deep models running against egocentric first-person images as are captured by wearable cameras like DeepEye. To our knowledge, our work along with [15] has evaluated the accuracy of these techniques on such image workloads. We reported that the inference accuracies for all models drop when they are applied on egocentric images. These reported accuracy numbers are expected to improve when techniques such as fine-tuning are applied to these models using the training images captured by DeepEye.

Privacy Concerns. Like any wearable camera, with DeepEye there are significant potential privacy concerns that must be respected. For this reason DeepEye never retains any images at any stage. Only inference results are kept. This is quite unlike most wearables that perform heavy image processing that will most of-

ten rely on the cloud. Users also benefit from a very simple mental model of where their data is and is not. They can assure themselves that data never leaves DeepEye unless they install an application that does so.

7. RELATED WORK

DeepEye follows in a line of wearable and mobile vision systems that have been prototyped for decades. Relatively recent examples include SenseCam [1], Gabriel [2] and ThirdEye [3]. A central difference to these systems is that DeepEye performs image processing using deep learning models; this is significant because this brings state-of-the-art models for various image task to a wearable platform, rather either simplified alternative learning algorithms executing locally or complex models running remotely. While few mobile systems (such as smartphones [42, 43]) and virtually no wearable systems (i.e., those with a smaller form factor) use a single deep learning for local processing, in contrast DeepEye integrates multiple concurrently executing models. DeepEye explores an area that is only just emerging (cloud-free deep learning for mobile vision), and more than that leaps beyond existing systems with single models to study what we believe will be a important upcoming step; specifically, how these systems can cope with (and benefit from) multiple executing deep learning models.

More importantly, DeepEye contributes significantly as a enabler of experiments to study these issues because all hardware and software will be open-sourced. In the area of mobile vision, perhaps the central systems question has been how these systems can be made to be energy (and more broadly resource) efficient [12, 14]. The DeepEye wearable and its cloud-free focus allows us to further understand when and if the cloud should be incorporated especially in relation to multiple deep model scenarios. Few studies of this type currently exist into the performance of deep models on mobile hardware; those that do exist (such as [28]) have not yet examined the multi-model problem, and more importantly have not made the type of observations for future optimizations in this area that we have done in this paper (see earlier section). SDKs for using mobile GPUs and deep models are starting to arrive (such as [11]). In contrast, other directions being take start at a more algorithmic level and consider how the models themselves can be different towards being more efficient [13, 44]. But interestingly most of the understanding of mobile and optimized deep learning currently exists in commercial settings (such as [42]) although software approaches to cloud offloading [9] and device-side acceleration [45] are arriving. The experiments we report here are some of the first to focus specifically on large deep models designed for image processing on a wearable. Examples of deep learning on wearables form-factors are very rare up to this point and have not considered the CNN architecture needed for images; in comparison work like [46] and [47] consider LSTM and RBMs respectively.

Finally, a key part of the future of deep learning on mobiles and wearables undoubtedly will include consideration of custom hardware. Such hardware options are evolving in research [48][39] and in industry [38]; DeepEye contributes to this examination by exploring the limit of conventional hardware through the combination of a conventional processor and custom supporting daughter board. Interesting ideas in co-design involving low-level ML components and sennsor array are also emerging towards more efficicent vision processing systems [10]. But because DeepEye leverages a general-purpose hardware its observations will more easily translate into other more popular SoCs, than deep learning specific hardware options. Furthermore, the observations for rethinking and optimizing the software-based inference pipeline is just as needed as any hardware innovations because, as we highlighted here, many optimization opportunities exist within the inference algorithms available.

8. CONCLUSION

In this paper, we empirically study two key areas pertaining to mobile vision systems moving forward. First, the raw feasibility, and overall system performance, of wearable form-factor devices to meet the needs of cloud-free deep learning-based vision processing of periodically captured images. We study this within the context of a powerful conventional small form-factor processor, and purpose-built daughter board, under a real-world image workload of ≈4000 images collected from a user trial with actual commercial wearable device. Second, we focus on a key aspect of execution support for these models – namely, the challenges presented by simultaneous inference of multiple models on the same image. This will be a building block behavior of vision systems as most applications require multiple types of analysis (i.e., multiple models) to be performed on a single image. We highlighted the core bottlenecks with executing multiple deep models on constrained devices, and proposed methods to manage these issues in the inference process when concurrent images are processed. We believe our techniques will be of broad interest for mobile systems and vision researchers at large, and will lead to more research in this space.

9. REFERENCES

[1] S. Hodges, L. Williams, E. Berry, S. Izadi, J. Srinivasan, A. Butler, G. Smyth, N. Kapur, and K. Wood, "Sensecam: A retrospective memory aid," in *UbiComp 2006: Ubiquitous Computing*. Springer, 2006, pp. 177–193.

[2] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 68–81. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594383

[3] S. Rallapalli, A. Ganesan, K. Chintalapudi, V. N. Padmanabhan, and L. Qiu, "Enabling physical analytics in retail stores using smart glasses," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '14. New York, NY, USA: ACM, 2014, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/2639108.2639126

[4] "Google Glass," https://developers.google.com/glass/distribute/glass-at-work.

[5] "Narrative Clip 1," http://getnarrative.com/narrative-clip-1.

[6] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[7] T. Weyand, I. Kostrikov, and J. Philbin, "Planet - photo geolocation with convolutional neural networks," *CoRR*, vol. abs/1602.05314, 2016. [Online]. Available: http://arxiv.org/abs/1602.05314

[8] A. Khosla, A. S. Raju, A. Torralba, and A. Oliva, "Understanding and predicting image memorability at a large scale," in *International Conference on Computer Vision (ICCV)*, 2015.

[9] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An execution framework for deep neural networks on resource-constrained devicesg," in

Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, 2016.

[10] M. Philipose, "Efficient object detection via adaptive online selection of sensor-array elements," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, ser. AAAI'14. AAAI Press, 2014, pp. 2817–2823. [Online]. Available: http://dl.acm.org/citation.cfm?id=2892753.2892942

[11] L. N. Huynh, R. K. Balan, and Y. Lee, "Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices," in *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, ser. WearSys '16. New York, NY, USA: ACM, 2016, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2935643.2935650

[12] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13. New York, NY, USA: ACM, 2013, pp. 69–82. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464448

[13] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy, "Fast video classification via adaptive cascading of deep models," *arXiv preprint arXiv:1611.06453*, 2016.

[14] P. Bahl, M. Philipose, and L. Zhong, "Vision: cloud-powered sight for all: showing the cloud what you see," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 53–60.

[15] D. Castro, S. Hickson, V. Bettadapura, E. Thomaz, G. Abowd, H. Christensen, and I. Essa, "Predicting daily activities from egocentric images using deep learning," in *Proceedings of the 2015 ACM International Symposium on Wearable Computers*, ser. ISWC '15. New York, NY, USA: ACM, 2015, pp. 75–82. [Online]. Available: http://doi.acm.org/10.1145/2802083.2808398

[16] P. Kelly, A. Doherty, E. Berry, S. Hodges, A. M. Batterham, and C. Foster, "Can we use digital life-log images to investigate active and sedentary travel behaviour? results from a pilot study," *Int J Behav Nutr Phys Act*, vol. 8, no. 44, p. 44, 2011.

[17] A. Grimes, M. Bednar, J. D. Bolter, and R. E. Grinter, "Eatwell: sharing nutrition-related memories in a low-income community," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*. ACM, 2008, pp. 87–96.

[18] R. Sarvas and D. M. Frohlich, *From snapshots to social media-the changing picture of domestic photography*. Springer Science & Business Media, 2011.

[19] I. Li, A. K. Dey, and J. Forlizzi, "Understanding my data, myself: supporting self-reflection with ubicomp technologies," in *Proceedings of the 13th international conference on Ubiquitous computing*. ACM, 2011, pp. 405–414.

[20] H. Pirsiavash and D. Ramanan, "Detecting activities of daily living in first-person camera views," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 2847–2854.

[21] A. J. Sellen, A. Fogg, M. Aitken, S. Hodges, C. Rother, and K. Wood, "Do life-logging technologies support memory for the past?: an experimental study using sensecam," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 81–90.

[22] L. Deng and D. Yu, "Deep learning: Methods and applications," Tech. Rep. MSR-TR-2014-21, January 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=209355

[23] M. Dimiccoli, M. Bolaños, E. Talavera, M. Aghaei, S. G. Nikolov, and P. Radeva, "Sr-clustering: Semantic regularized clustering for egocentric photo streams segmentation," *arXiv preprint arXiv:1512.07143*, 2015.

[24] "Autographer," http://getnarrative.com/narrative-clip-1.

[25] N. D. Lane, S. Bhattacharya, C. Forlivesi, P. Georgiev, L. Jiao, L. Qendro, , and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN 2016*.

[26] "Perceptual Hashing," http://www.phash.org/.

[27] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2016.

[28] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices," in *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, ser. IoT-App '15. New York, NY, USA: ACM, 2015, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2820975.2820980

[29] "BLAS library," http://www.netlib.org/blas/.

[30] S. S. Farfade, M. J. Saberian, and L.-J. Li, "Multi-view face detection using deep convolutional neural networks," in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*. ACM, 2015, pp. 643–650.

[31] G. Levi and T. Hassner, "Age and gender classification using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2015, pp. 34–42.

[32] G. Levi and T. Hasner, "Emotion recognition in the wild via convolutional neural networks and mapped binary patterns," in *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*, 2015, pp. 503–510.

[33] J. Zhang, S. Ma, M. Sameki, S. Sclaroff, M. Betke, Z. Lin, X. Shen, B. Price, and R. Mech, "Salient object subitizing," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 4045–4054.

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[35] A. Khosla, A. S. Raju, A. Torralba, and A. Oliva, "Understanding and predicting image memorability at a large scale," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2390–2398.

[36] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in neural information processing systems*, 2014, pp. 487–495.

[37] N. Lane, S. Bhattacharya, A. Mathur, C. Forlivesi, and F. Kawsar, "Dxtk: Enabling resource-efficient deep learning on mobile and embedded devices with the deepx toolkit," in *Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services*, ser. MobiCASE'16. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 98–107.

[Online]. Available:
https://doi.org/10.4108/eai.30-11-2016.2267463

[38] "Movidius," http://www.movidius.com/.

[39] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[40] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition," in *INTERSPEECH*, 2013, pp. 2365–2369.

[41] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 1135–1143.

[42] "How Google Translate squeezes deep learning onto a phone," http://googleresearch.blogspot.co.uk/2015/07/how-google-translate-squeezes-deep.html.

[43] "Happy Halloween! Baidu Research Introduces FaceYou," http://uk.reuters.com/article/idUKnMKWYbQbJa+1ea+MKW20151029.

[44] S. Venkataramani, V. Bahl, X.-S. Hua, J. Liu, J. Li, M. Phillipose, B. Priyantha, and M. Shoaib, "Sapphire: an always-on context-aware computer vision system for portable devices," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1491–1496.

[45] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, 2016.

[46] T. Beltramelli and S. Risi, "Deep-spying: Spying using smartwatch and deep learning," *CoRR*, vol. abs/1512.05616, 2015. [Online]. Available: http://arxiv.org/abs/1512.05616

[47] S. Bhattacharya and N. D. Lane, "From smart to deep: Robust activity recognition on smartwatches using deep learning," in *Proceedings of the Second Workshop on Sensing Systems and Applications Using Wrist Worn Smart Devices*, 2016.

[48] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *International Symposium on Computer Architecture (ISCA)*, 2016.