Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables

Sourav Bhattacharya[§] and Nicholas D. Lane[§,†]
[§]Nokia Bell Labs, [†]University College London

ABSTRACT

Deep learning has revolutionized the way sensor data are analyzed and interpreted. The accuracy gains these approaches offer make them attractive for the next generation of mobile, wearable and embedded sensory applications. However, state-of-the-art deep learning algorithms typically require a significant amount of device and processor resources, even just for the inference stages that are used to discriminate high-level classes from low-level data. The limited availability of memory, computation, and energy on mobile and embedded platforms thus pose a significant challenge to the adoption of these powerful learning techniques. In this paper, we propose SparseSep, a new approach that leverages the sparsification of fully connected layers and separation of convolutional kernels to reduce the resource requirements of popular deep learning algorithms. As a result, SparseSep allows large-scale DNNs and CNNs to run efficiently on mobile and embedded hardware with only minimal impact on inference accuracy. We experiment using SparseSep across a variety of common processors such as the Qualcomm Snapdragon 400, ARM Cortex M0 and M3, and Nvidia Tegra K1, and show that it allows inference for various deep models to execute more efficiently; for example, on average requiring 11.3 times less memory and running 13.3 times faster on these representative platforms.

CCS Concepts

•Computing methodologies → Machine learning; Neural networks; •Computer systems organization → Embedded software;

Keywords

Wearable computing; deep learning; sparse coding; weight factorization

1. INTRODUCTION

Recognizing contextual signals and the everyday activity of users from raw sensor data is a core enabler for mobile and wearable applications. By monitoring user actions (via speech, ambient audio, motion) and context using a variety of sensing modalities, mobile developers are able to provide both enhanced, and brand new, application features. While sensor-related applications and systems are still maturing, and are highly diverse, a notable characteristic is their reliance on making a wide-variety of sensor inferences.

Accurately extracting context and activity information from noisy mobile sensor data remains an unsolved problem. Because the real world is highly complex, unpredictable and constantly changing, it often causes confusion to the machine learning and signal processing algorithms used by mobile devices. One of the most promising directions today in overcoming such challenges is *deep learning* [1, 2]. Developments in this particular field of machine learning have caused the approaches and algorithms used in even mature sensing tasks to be completely changed (e.g., speech [3] and face [4] recognition). The study of deep learning usage for mobile applications is in its early stages (e.g., [5, 6, 7, 8]), but with promising initial results.

While deep learning offers important benefits to robust modeling, its integration into mobiles and wearables is complicated by the sizable system resource requirements these algorithms introduce. Barriers exist in the form of memory, computation and energy; these collectively prevent most deep models from executing directly on mobile hardware. Consequently, existing examples of deep learning for smartphones (e.g., speech recognition) remain largely cloud-assisted. A number of negative side-effects of this: first, inference execution becomes coupled to fluctuating and unpredictable network quality (e.g., latency, throughput); but more importantly it exposes users to privacy dangers [9] as sensitive data (e.g., audio) is processed off-device by a third party.

Allowing broader *device-centric* deep learning classification and prediction will need the development of brand-new techniques for optimized resource sensitive execution. Up to this point, the machine learning community has made excellent progress in training-time optimizations and is only now beginning to consider how these ideas transfer to inference-time. Currently, most knowledge of deep learning algorithm behavior on constrained devices is largely limited to one-off task-specific experiences (e.g., [10, 11]). These systems are limited however is providing examples and evidence that local execution is feasible, although they do provide some insights for ways forward. What is required however is a deeper study of these issues with an aim towards the development of techniques like off-line model optimization and runtime execution environments to match the resources (e.g., memory, computation and energy) present on edge devices like wearables and mobile phones.

In this work, we make an significant progress into the development of such algorithms and software by developing a sparse coding- and convolution kernel separation-based approach to optimizing deep learning model layers. This framework – SparseSep – includes: (1) a compiler, Layer Compression Compiler (LCC), in which unchanged deep models are inserted and then optimized; (2) a runtime framework, Sparse Inference Runtime (SIR), that is able to exploit the transformation of the model and realize radical reductions in computation, energy and memory usage; and (3) a separator, Convolution Separation Runtime (CSR), that significantly reduces convolution operations. SparseSep techniques can allow a developer to adopt existing off-the-shelf

deep models and scale their processor behavior such as, acceptable accuracy reduction and device limits, e.g., memory and necessary execution time.

The core concept of this work is the hypothesis that computational and space complexity of the deep learning models can be significantly improved through the sparse representation of key layers and separation of convolution layers. Deep models often have millions of parameters spread throughout a number of hidden layers that capture the robust representations of the data. By using theory from sparse dictionary learning we investigate how the originally complex synaptic weight matrix can be captured in much smaller matrices that require less computational and memory resources. Critically, such theory affords the ability of these sparsified layers to be faithful to the originals with theoretical bounds on important aspects such as, reconstruction error. This is the first time this approach has been used.

Our experiments include both DNNs and CNNs, the most popular forms of deep learning today. Tests span both audio classification tasks (ambient scene analysis and speaker identification) that are common in the mobile sensing systems; along with image tasks (object recognition) seen in mobile vision devices like Google Glass. We find that across a range of experiments and devices SparseSep can allow deep models to execute using (on an average) only 26% of the original energy while only sacrificing approximately up to 5% of the accuracy of these models. Specific examples include the Snapdragon 400 processor running a deep learning model for speaker identification with a 4.1 times improvement in execution time, and a 17.6 times reduction in memory. Furthermore, we benchmark this deep learning version of speaker identification and find, as expected, that the deep model is much more robust than models conventionally used (such as random forests). Most important of all, we examine device restrictions found on other common processors like the Cortex M3 equipped with 32 KB of RAM. Not surprisingly we find these processors can not support any form of deep learning model tested (due to restrictions to computation and/or memory) – *until* we apply the SparseSep process.

The key scientific contributions of this research are:

- We propose, for the first time, a sparse coding-based approach to the optimization of deep learning inference execution. We propose the use of convolution kernel separation technique to minimize overall computations of CNNs on resource constrained platforms.
- To our knowledge, this work is the first to demonstrate very deep learning models (many layer DNNs and CNNs) executing on severely constrained wearable hardware with acceptable levels of performance (energy efficiency, computation times).
- We design and implement a prototype that realizes our approach to sparse dictionary learning and kernel separation into deep learning model representation and inference execution. We implement necessary runtime components for 4 embedded and mobile processor platforms.
- We experiment with four different CNN and DNN models under large audio and image datasets. We demonstrate gains of the order of 11.3× improvements in memory and 13.3× in execution time under multiple experiment configurations, while only suffering accuracy loss of ≈ 5%.

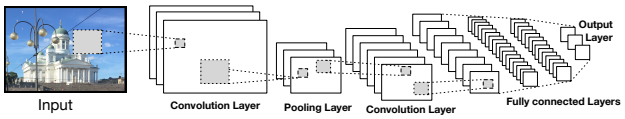


Figure 1: A CNN mixes convolutional and feed-forward layers

2. BACKGROUND

Popular deep learning architectures, such as Restricted Boltzmann Machines and Deep Belief Networks, share a common architecture. Often, they are collectively referred to as Deep Neural Networks. Typically, a DNN contains a number of fully-connected layers, where each layer is composed of a collection of nodes. Sensor measurements (e.g., audio, images) are fed to the first layer (the input layer). The final layer, also known as the output layer, corresponds to inference classes with nodes capturing individual inference categories (e.g., music or cat). Layers in between the input and the output layer are referred to as hidden layers. The degree of influence of units between layers vary on a pairwise basis determined by a weight value. Together with the synaptic connections and inherent non-linearity, the hidden layers transform raw data applied to the input layer into the prediction classes captured in the output layer.

DNN-based inferencing follows a feed-forward algorithm that operates on sensor data segments in isolation. The algorithm starts at the input layer and moves layer wise sequentially, while updating the activation states of all nodes one by one. The process finishes at the output layer when all nodes have been updated. Finally, the inferred class is identified as the class corresponding to the output layer node with the greatest state value.

CNNs are another popular class of deep models that share architectural similarities to DNNs. As presented in Figure 1, a CNN model contains one or more convolutional layers, pooling or sub-sampling layers, and fully connected layers (equivalent to those used in DNNs). The objective of these layers is to extract simple representations from the input data, and then converting the representation into more complex representations at much coarser resolutions within the subsequent layers. For instance, first convolutional filters (with small kernel width) are applied to the input data to capture local data properties. Next, max or min pooling is applied to make the representations invariant to translations. Pooling operations can also be seen as a form of dimensionality reduction. Lastly, fully connected layers (i.e., a DNN) help a CNN to make predictions.

A CNN follows a sequential approach, as in DNNs, to generate isolated prediction at a time. Often in CNN-based predictions, sensor data is first vectorized into two dimensions. Next, data is passed through a series of convolution, pooling and non-linear layers. The purpose of the convolution and pooling layers can be viewed as that of feature extractor before the fully connected layers are engaged. Inference then proceeds exactly as previously described for DNNs until ultimately a classification is reached.

Contrary to the shallow learning-based models, deep learning models are usually big an often contains more than million parameters. High parameter space improves the capacity of these models and they often outperform prior shallow models in terms of model generalization performances. However, the accuracy gains come at the expense of high energy

and memory costs. Although, high end wearables containing GPU, e.g., NVIDIA Tegra K1, can efficiently run deep models [12], the high resource demands make deep learning models unattractive for low end wearables. In this paper we explore sparse factorizations and convolutional kernel separations to optimize the resource demands of deep models, while maintaining the functional properties of the models.

3. DESIGN AND OPERATION

Beginning with this section, and spanning the following two, we detail the design and algorithms of SparseSep.

3.1 Design Goals

SparseSep is shaped on the following objectives.

- **No Re-training.** The training of a large deep model is the most time consuming and computationally demanding task. For example, a large model such as GoogleNet is trained using thousands of CPU cores [13], which is beyond the current capabilities of a single wearable device. In this work, we mainly focus on the inference cycle of a deep model and perform no training on the resource-constrained devices. The training process also requires a very large training dataset, often inaccessible to the developers [14]. Thus new techniques are needed to compress popular cloud-scale deep learning models to run on wearable and IoT grade hardware gracefully.
- **No Cloud Offloading.** As noted in §1, offloading the execution of portions of deep models can result in leaking sensitive sensor data. By keeping inference completely local, user and applications have greater privacy protection as the data or any intermediate results never leave the device.
- **Target Low-resource Platforms.** Even high-end mobile processors (such as the Tegra K1 [15]) still require careful resource use, when executing deep learning models. But in this class of processors, the gap in resources is closing. However, for low-energy highly portable wearable processors that lack GPUs or have only a few MBs of RAM (e.g., ARM Cortex M3 [16]), local execution of deep models remains impractical. For this reason, SparseSep turns to new ideas like the use of sparsification of weights and kernel separation, in search of the leaps in resource efficiency required to make these low-end processors viable.
- **Minimize Model Changes.** Deep models must undergo some degree of change to enable their operation on wearable hardware. However, a core tenet of SparseSep is to minimize the extent of such modifications and remain functionally faithful to the initial model architecture. For this reason, we frame the problem as one of deep model compression (originally formulated by the machine learning community), where model layer arrangements remain unchanged and only per-layer connections are changed through the insertion of additional summarizing layers. Thus, the degree of changes made by SparseSep is a key metric that is minimized during model processing.
- **Adopt Principled Approaches.** Ad-hoc methods to alter a deep model – such as '*specializing*' a model to recognize a smaller set of activities/contexts, or changing layer/unit parameters to generate a desired resource consumption profile – are dangerous as they violate the domain experience of the modeling experts. Methods like sparse coding [17] and model compression [18] are supported by theoretical analysis [19]. Assessing if a model can be altered solely by changes in the accuracy metric can be dangerous and can potentially hurt, for example, its ability to generalize.

3.2 Overview

We now briefly outline the core approach of SparseSep to optimize the architecture of large deep learning models so that they meet the constraints of target wearable devices. In §4 we provide the necessary theory and algorithms of this process, but we begin here with the key ideas.

The inference pipeline of a deep learning model is dominated by a series of matrix computations, especially multiplications, and convolutions. Attempts have been made to optimize the total number of computations by low-rank factorizing of the weight matrix or decomposing convolutional kernels into separable filters in an ad-hoc manner. Both weight factorization and kernel separation, however, require modification in the architecture of the model by inserting a new layer and updating weight components (see §4.1 and §4.4). Although, counter-intuitive, the insertion of a new layer only achieves computational efficiency under certain conditions, which depends on, e.g., the size of the newly inserted layer, the size of the original weight matrix, and the size of convolutional kernels. In §4.1, §4.2 and §4.4 we derive and show the conditions under which computational and memory efficiencies can be achieved.

In this paper, we postulate that the computational and space efficiency of the deep learning models can be further improved by adding sparsity constraints to the factorization process. Accordingly, we propose a sparse dictionary learning approach to enforcing a sparse factorization of the weight matrix (see §4.3). In §5.2 we show that under specific sparsity conditions the resource scalability of the proposed approach is significantly better than existing approaches.

The weight factorization approach significantly reduces the memory footprint of both DNN and CNN models by optimizing the parameter space of the fully connected layers. The factorization also helps to reduce the overall number of operations needed and improves the inference time. However, the inference time improvement due to factorization is much more pronounced for DNNs than CNNs. This is primarily due to the fact that a major portion of the CNN-based inference time (often over 95%) is spent on performing convolution operations [12, 20], where the layer factorization technique has no influence. To overcome this limitation, we also propose a runtime convolution kernel separation technique that optimizes the convolution operations to reduce overall inference time and energy expenditure of CNN models. Together with the weight factorization technique, the convolution optimization reduces both memory and energy footprints of cloud-scale CNNs. In §4.4 details of the runtime convolution optimization are provided.

3.3 Implementation and Operation

To examine the SparseSep techniques, we prototype three software components: Layer Compression Compiler, Sparse Inference Runtime and Convolution Separation Runtime; these are briefly described below, and shown in Figure 2.

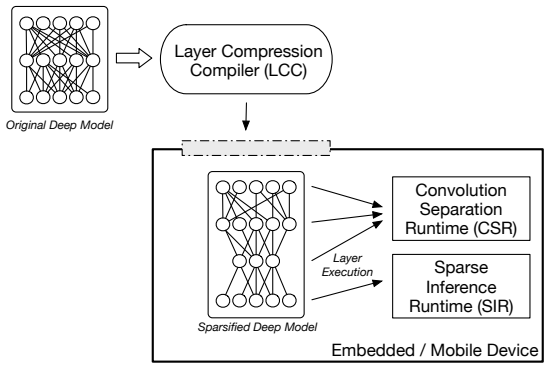


Figure 2: SparseSep Framework and Operation

Layer Compression Compiler (LCC). Prior to a deep model being used on a wearable device, it is processed to apply sufficient compression to satisfy device constraints. No machine learning expertise is required, developers only specify constraints of the target hardware to LCC including the memory limits and computational constraints in terms of execution time. LCC automatically applies the sparse coding-based factorizations described above throughout the deep model. The objective is to determine a set of insertion positions and compression degrees that match the required resource constraints, while also minimizing any loss of accuracy. In §4, this procedure is described in detail as well as how the search for the optimal configuration of compression layers is performed efficiently.

Sparse Inference Runtime (SIR). To maximize the benefit of the model produced by LCC, modifications are required to the fully connected layers of DNNs/CNNs. First, compaction layers are k-sparse and it requires a series of key modifications to be made on the standard feed-forward style inference approach across the modified model. The modification heavily utilizes the inherent sparse structure of the generated weight matrices to gain computational efficiency. Second, to assist those platforms that are constrained by memory severely, layers are executed one by one. For instance, only the matrices related to the specific two layers being executed are loaded into the memory and computed on. The decision to do this for a layer pair is taken by the compiler and done conservatively if it is expected to introduce unwanted delays in the inference time due to the overhead of additional load times.

Convolution Separation Runtime (CSR). To further minimize the inference time of a CNN model, after modified by the LCC, convolution layer modifications are needed. The time constraint provided by a developer is again used to select a set of convolutional layers and their compression levels are determined to meet the overall resource goals. In the case of severe memory unavailability, strategies similar to SIR are employed (see §4.4). The goal here is to keep the functional behavior of the modified model as close as possible to the original unmodified model.

4. ALGORITHMIC FOUNDATIONS

The core components of SparseSep rely heavily on a number of algorithms to select, compress, and optimize both fully connected and convolution layers of deep models. In the following we begin by briefly explaining the computational requirements of typical deep models (e.g., DNN and CNN). We then provide intuitions for optimizations, describe in detail the sparse weight factorization and the convolutional separation approaches employed by SparseSep, and highlight the necessary conditions and benefits of the techniques on memory footprint and computational efficiency.

4.1 Deep Model Computations

The inference task of DNNs can be summarized as a series of matrix multiplications, vector additions and evaluations of non-linear functions. For example, the output $\boldsymbol{o}$ of neural network with a single hidden layer can be computed as:

$$\boldsymbol{o} = \text{SoftMax}\left(\boldsymbol{b}^2 + \boldsymbol{W}^2 \cdot f(\boldsymbol{b}^1 + \boldsymbol{W}^1 \cdot \boldsymbol{x})\right), \tag{1}$$

where $\boldsymbol{x}$ is the input vector, $f(\cdot)$ is the non-liner function (e.g., *sigmoid*), $\boldsymbol{b}^i$ is the bias vector and $\boldsymbol{W}^i$ is the weight matrix associated with layer i. The matrix operations can be efficiently computed using, e.g., a GPU, while applying new vectorization techniques [21]. Development of computational optimization is complementary to the development of efficient hardware, as it often enables running a deep model more efficiently on the new platform. However, GPUs are seldom available on wearable platforms due to their large energy footprints. We address both memory and computation optimization tasks for fully connected layers by drawing inspirations from the well-known *matrix chain multiplication* problem [22].

In case of a CNN with one convolution, one pooling and one hidden layer, the output $\boldsymbol{o}$ can be computed as:

$$\boldsymbol{o} = \text{SoftMax}\left(\boldsymbol{b}^2 + \boldsymbol{W}^2 \cdot \text{maxpool}\left[\boldsymbol{M}\right]\right), \tag{2}$$

where, $\boldsymbol{M}$ is the *feature map* computed from the $2D$ input $\boldsymbol{x}$ as:

$$\boldsymbol{M}_j = f\left(\sum_c \boldsymbol{x}^c * \mathcal{K}_j^c + \boldsymbol{b}_j^1\right), \tag{3}$$

where, c represents the index over channels and $\mathcal{K}$ is the set of learnable convolution kernels.

4.2 Weight Factorization

In case of a fully connected layer, updating states of all nodes requires evaluating the product:

$$\boldsymbol{W}^L \cdot \boldsymbol{x}^L \tag{4}$$

where, $\boldsymbol{x}^L \in \mathbb{R}^n$ is the state of nodes in the previous layer and $\boldsymbol{W}^L \in \mathbb{R}^{m \times n}$ is the matrix representing all the connections between layer L and $L+1$. Now, the basic idea in decreasing the number of required computations is to replace the weight matrix $\boldsymbol{W}^L$ with a product of two different matrices, i.e.,

$$\boldsymbol{W}^L = \boldsymbol{U} \cdot \boldsymbol{V} \tag{5}$$

where $\boldsymbol{U} \in \mathbb{R}^{m \times k}$, $\boldsymbol{V} \in \mathbb{R}^{k \times n}$, such that the total number of computations needed to compute $\boldsymbol{U} \cdot (\boldsymbol{V} \cdot \boldsymbol{x}^L)$ becomes smaller than the original multiplication (see Equation 4). In other words, computational efficiency can be achieved, when the total number of multiplications in $\boldsymbol{U} \cdot (\boldsymbol{V} \cdot \boldsymbol{x}^L)$ is smaller than the number of multiplications in $\boldsymbol{W}^L \cdot \boldsymbol{x}^L$, i.e.:

$$k \cdot n \cdot 1 + m \cdot k \cdot 1 \quad < \quad m \cdot n \cdot 1 \tag{6}$$

$$\implies k \quad < \quad \frac{m \cdot n}{m + n} \tag{7}$$

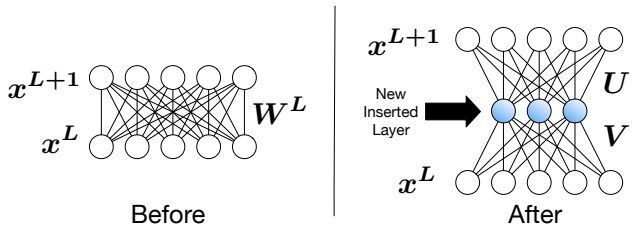


Figure 3: Layer insertion to achieve computational efficiency

Hence, the above equation gives us a rule for selecting the dimensionality of $\boldsymbol{U}$ and $\boldsymbol{V}$ matrices to achieve computational efficiency. The computational gain due to the factorization of weight matrix $\boldsymbol{W}^L$ can be easily implemented by first removing the connections $\boldsymbol{W}^L$, followed by introducing a new layer with k nodes[1] and finally updating new connections to the adjacent layers with weights $\boldsymbol{V}$ and $\boldsymbol{U}$ respectively. Figure 3 illustrates the architectural modifications needed to allow weight factorization-based optimizations.

Weight factorization also brings memory benefits. For example, before any architectural modifications, the total number of parameters needed to compute $\boldsymbol{x}^{L+1}$ is $m \cdot n + m$, i.e., all the elements of matrix $\boldsymbol{W}^L$ and the biases of layer $L+1$. Once the weight matrix is factorized and replaced, $n \cdot k + k \cdot m + m$ parameters are required for evaluating $\boldsymbol{x}^{L+1}$. Interestingly, a decrease in model size can be achieved if the following inequality holds:

$$n \cdot k + k \cdot m + m < m \cdot n + m, \tag{8}$$

The above inequality simplifies to the same requirement as identified in Equation 7. The space gain S_g due to layer modification can be defined as:

$$S_g = \frac{m \cdot n + m}{n \cdot k + k \cdot m + m} \tag{9}$$

Ideally, we seek a factorization resulting in $S_g \gg 1$.

4.3 Weight Reconstruction

Note that, under loss-less factorization, i.e., no error in the reconstruction of $\boldsymbol{W}^L$, the modified architecture of the deep model stays functionally equivalent to the original. However, often arbitrary factorizations of large matrices introduce reconstruction errors and thus the new constructed model deviates from the original model and affects the oveall performance accuracy. Thus, care should be taken to keep the reconstruction error (e.g., L_2-norm) as small as possible, i.e.:

$$||\boldsymbol{W}^L - \boldsymbol{U} \cdot \boldsymbol{V}||_2^2 \approx \boldsymbol{0} \tag{10}$$

Previously, optimization of DNN models have been attempted using the well established *Singular Value Decomposition* (SVD) approach [23, 24]. Under SVD, the weight matrix can be efficiently factorized as:

$$\boldsymbol{W}^L_{m \times n} = \boldsymbol{X}_{m \times m} \cdot \boldsymbol{\Sigma}_{m \times n} \cdot \boldsymbol{N}^T_{n \times n}, \tag{11}$$

where, $\boldsymbol{\Sigma}_{m \times n}$ is a rectangular diagonal matrix containing *singular values* of $\boldsymbol{W}^L_{m \times n}$ as the diagonal elements. To gain computational efficiency the weight matrix can be approximated well by keeping k highest singular values, i.e.:

$$\boldsymbol{W}^L_{m \times n} \approx \boldsymbol{X}_{m \times k} \cdot \boldsymbol{\Sigma}_{k \times k} \cdot \boldsymbol{N}^T_{k \times n} \tag{12}$$

Now, the architecture of a fully connected layer of a deep model can be modified by replacing $\boldsymbol{W}^L$ with $\boldsymbol{U} = \boldsymbol{X}_{m \times k}$ and $\boldsymbol{V} = \boldsymbol{\Sigma}_{k \times k} \cdot \boldsymbol{N}^T_{k \times n}$ (see Figure 3).

4.4 Sparse Coding-based Factorization

In the following, we propose a novel technique of improving the memory and computational gains over the SVD-based factorizations (for a given k). These gains are essential for efficiently running deep models on low-end wearables. The sparsity requirement also makes SparseSep novel from our previously published DeepX system [14].

4.4.1 Dictionary Learning

The basic idea is to come up with a sparse factorization of the weight matrix $\boldsymbol{W}^L$. In other words, if either of the $\boldsymbol{U}$ or $\boldsymbol{V}$ matrices can be made sparse, further space savings can be achieved. Furthermore, sparse matrix multiplication also helps to improve overall computational time. In this work, we introduce the use of sparse coding-based factorization technique of fully connected layer weights to achieve very low memory and computational footprint.

The sparse matrix factorization problem can be formulated as a sparse dictionary learning problem, where a dictionary $\mathcal{B} = \{\boldsymbol{\beta}\}_{i=1}^k$ (with $\boldsymbol{\beta}_i \in \mathbb{R}^m$) is learned from the weights $\boldsymbol{W}^L$ of a fully connected layer of a deep model using an unsupervised algorithm. Sparse coding approximates an input $\boldsymbol{w}_i \in \mathbb{R}^m$, e.g., a column of $\boldsymbol{W}^L$, as a sparse linear combination of basis vectors $\boldsymbol{\beta}$ from the dictionary $\mathcal{B}$ [25, 26, 27], i.e.:

$$\boldsymbol{w}_i = \sum_{j=1}^{k} a_j^i \cdot \boldsymbol{\beta}_j, \tag{13}$$

where, $\boldsymbol{a}^i$ is a sparse vector, i.e., majority of its elements are 0. A large number of dictionary learning algorithms have been proposed in the literature and in this paper we use the K-SVD algorithm [28] to learn a dictionary comprising of k basis vectors. The K-SVD algorithm learns a dictionary by solving the following minimization problem:

$$\min_{\boldsymbol{\mathcal{B}}, \boldsymbol{A}} ||\boldsymbol{W}^L - \boldsymbol{\mathcal{B}} \cdot \boldsymbol{A}||_2^2 \quad s.t.\ \forall i\ ||\boldsymbol{a}^i||_0 \leq K \tag{14}$$

where $\boldsymbol{A} \in \mathbb{R}^{k \times n}$ is the sparse code of the weight matrix $\boldsymbol{W}^L$ under the dictionary $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{m \times k}$ and K is the sparsity constraint factor, i.e., each signal is reconstructed with fewer than K basis vectors. Once the dictionary $\boldsymbol{\mathcal{B}}$ and the sparse code $\boldsymbol{A}$ is learned, we obtain the sparse factorization of $\boldsymbol{W}^L$ as:

$$\boldsymbol{W}^L \approx \boldsymbol{\mathcal{B}} \cdot \boldsymbol{A} \tag{15}$$

4.4.2 Architecture Modification

To gain computational and space efficiency, we interpose a new layer similarly as described in §4.2, while setting $\boldsymbol{U} = \boldsymbol{\mathcal{B}}$ and $\boldsymbol{V} = \boldsymbol{A}$ (see Figure 3). DNN and CNN models often have more than one fully connected layers and thus the sparse factorization technique can be applied on all of these layers to improve overall inference efficiency.

[1] We also set the bias of each node to 0 and no no-linearity function is added.

4.4.3 Quality of Code Book and Sparsity

The success of the sparse factorization of weights depends on the quality of the learned dictionary, its reconstruction quality and the sparsity of the generated code. However, as in the case with the SVD approach, the dictionary learning with sparsity constraint introduces reconstruction error and the modified model deviates from the original model in the parameter space. Deviation in the parameter space often adversely affect the performance quality of the model and thus care should be take to maintain the reconstruction error at low as possible. This can be achieved by introducing a hyper parameter α [27] that assigns a greater importance in minimizing the reconstruction error term than the sparsity requirement as given in Equation 14. In computer vision and audio recognition tasks, often over-complete dictionaries are learned for resilient feature learning. However, in line with our objective of achieving space and computational efficiency, we extract a compact representation of deep model parameters by learning a small dictionary.

4.4.4 Memory Gain

One of the main advantages of making sparse factorization of fully connected layers, over techniques using SVD, is its ability to reduce memory footprint significantly. For example, the space gain under sparse factorization becomes:

$$S_g^* = \frac{m \cdot n + m}{2 * nnz(\boldsymbol{A}) + k \cdot m + m}, \tag{16}$$

where, $nnz(\boldsymbol{A})$ counts the number of non-zero elements in matrix $\boldsymbol{A}$. The factor 2 is used to take into account storage for indices[2] of the non-zero elements of a sparse matrix. Thus, sparse factorization outperforms SVD in terms of memory by a factor of:

$$\frac{S_g^*}{S_g} = \frac{n \cdot k + k \cdot m + m}{2 * nnz(\boldsymbol{A}) + k \cdot m + m} \tag{17}$$

$$= 1 + \frac{n \cdot k - 2 * nnz(\boldsymbol{A})}{2 * nnz(\boldsymbol{A}) + k \cdot m + m} \tag{18}$$

Thus, for $nnz(\boldsymbol{A}) < n \cdot k/2$, $S_g^* > S_g$. Hence, we get a rule for selecting the variable K in Equation 14 suitably to reduce the bottle neck in memory.

4.4.5 Execution Pipeline

The main purpose of LCC is to bring sparse factorization as an offline tool for modifying large deep models. The feed-forward architecture of DNN allows a unique opportunity to apply a layer-wise partial inference scheme. Under this approach, layers of a DNN are sequentially loaded in the memory and their output is retained. Given the maximum memory, we can estimate the number of nodes k to use in the inserted layer for a pre-defined sparsity amount, e.g., 15%. In extreme cases, when two matrices can not simultaneously be loaded on the memory, simple tricks like divide and conquer approach to matrix multiplication can be employed that only require partial portions of the matrices in the memory. Partial multiplication of matrices, however, increases paging amount, which increases the overall inference time.

Contrary to the constraint on available memory, satisfying a given limit on the accuracy degradation is non-trivial.

[2]In case of contiguous memory allocation, one index is enough to identify elements in a matrix.

Algorithm 1 Satisfy Accuracy Constraint

1: **Input:** (i) $\mathcal{M}$: a DNN/CNN, (ii) $\mathcal{V}$: a validation dataset and (iii) A_{TH}: max allowed degradation in accuracy (e.g., 5%)
2: **Output:** (i) $\hat{\mathcal{M}}$: an optimized DNN/CNN
3: FCLayer := findAllFCLayers($\mathcal{M}$)
4: **for** $i := 1 : length$(FCLayer) **do**
5: $k_u := getUpperBound(FCLayer[i].\boldsymbol{W})$ ▷ Getting an estimate for k using Equation 7
6: $k_l := 1$
7: $k_c := k_l$
8: **while** True **do** ▷ Searching for suitable k in the range (k_l, k_u) using binary search
9: k_c :=updateBinarySerchParameter(k_c, k_l, k_u)
10: $\boldsymbol{U}, \boldsymbol{V} := sparseFactorize(FCLayer[i].\boldsymbol{W}, k_c)$
11: $newLayer := constructNewLayer(\boldsymbol{U}, \boldsymbol{V})$
12: $\hat{\mathcal{M}} := replaceLayer(FCLayer[i], newLayer)$
13: $D_A := getPerformanceDeviation(\hat{\mathcal{M}}, \mathcal{V})$
14: **if** $D_A < A_{TH}$ **then**
15: SaveModel($\hat{\mathcal{M}}$)
16: Break
17: **Return** readSavedModel()

This is because of the fact that there is no linear correlation between the reconstruction error and the model accuracy. Little variations in the weights can force a number of nodes to switch states, thereby potentially affecting the inference quality. To mitigate the problem we rely on a validation dataset provided along with the model and employ a search strategy to satisfy the accuracy degradation limit. Algorithm 1 provides an overview of the search procedure. Given the model, validation dataset and accuracy degradation bound, the algorithm searches for sparse factorization of one or more fully connected layers. For each layer, the algorithm follows a binary search procedure to estimate a suitable value for k and measures the accuracy degradation after applying sparse factorization of the weight matrix and replacing the layer as shown in Figure 3. If the accuracy bound is satisfied, this sparse factorization is accepted and the algorithm proceeds to the next fully connected layer.

Inference time of a DNN model, among other things, primarily depends on its parameter size and on the processing capabilities of the hardware. We take opportunistic advantage of Algorithm 1 and log all values of k used in individual layers (i.e., independent variables) and inference time (dependent variable) to build a simple regression model. The regression model can predict the execution time given parametric setting of k values in each layers. If the time prediction is higher than the given time constraint, we neglect saving the model. For example a simple conditional statement on time after line–14 in Algorithm 1 can be added to satisfy the time constraint.

4.5 Convolution Kernel Separation

In addition to the large memory requirement, the other bottleneck common in CNN-based inferencing (contrary to DNNs) is the massive amount of convolution operations. Weight factorization does not help to overcome this bottleneck. In the following we summarize techniques to mitigate this challenge.

4.5.1 Convolution Complexity

The time complexity of convolving a single channel $2D$-input $(H \times W)$ with a bank of N $d \times d$ filters is $\mathcal{O}(Nd^2HW)$. For large input image stacks with C channels, the runtime can be significantly large [29], e.g., $\mathcal{O}(CNd^2HW)$. One way to reduce the time complexity is by reducing the redundancy among different filters and filter channels [30]. Although, approximation techniques using linear combination of a smaller set of filters have been successfully attempted [29], in this work we exploit separable filter property to reduce the overall convolution operations.

4.5.2 Separable Filters

Let $\mathcal{K} \in \mathbb{R}^{N \times d \times d \times C}$ be the set of convolutional kernels (4D), here the goal is to find an approximation $\hat{\mathcal{K}}$, which can be decomposed as [31]:

$$\hat{\mathcal{K}}_n^c = \sum_{k=1}^{K} \mathcal{H}_n^k (\mathcal{V}_k^c)^T, \tag{19}$$

where, the parameter K controls the rank of the *horizontal filter* $\mathcal{H} \in \mathbb{R}^{N \times 1 \times d \times K}$ and the *vertical filter* $\mathcal{V} \in \mathbb{R}^{K \times d \times 1 \times C}$. Under this approximation scheme, the original convolution task of a single $3D$ filter[3] (indexed by n) becomes:

$$\begin{aligned} \mathcal{K}_n * \boldsymbol{x} &\approx \hat{\mathcal{K}}_n * \boldsymbol{x} \\ &= \sum_{c=1}^{C} \sum_{k=1}^{K} \mathcal{H}_n^k (\mathcal{V}_k^c)^T * \boldsymbol{x}^c \\ &= \sum_{k=1}^{K} \mathcal{H}_n^k * \left(\sum_{c=1}^{C} \mathcal{V}_k^c * \boldsymbol{x}^c \right) \end{aligned} \tag{20}$$

4.5.3 Architecture Modification

Both $\mathcal{H}$ and $\mathcal{V}$ filters can be learned from the pre-trained filter $\mathcal{K}$. In this work we adopt a deterministic SVD-based approximation algorithm, as given in [31], to estimate $\mathcal{H}, \mathcal{V}$. Interestingly, Equation 20 shows that the overall convolution task can be broken down into two sets of convolutions. First, the input $\boldsymbol{x}$ is convoluted with the vertical filter $\mathcal{V}$ to produce an intermediate feature map Z. Next, Z is convoluted with the horizontal filter $\mathcal{H}$ to generate the desired output. Thus, the original convolution layer of the CNN now can be replaced with two successive convolution layers with filters $\mathcal{V}$ and $\mathcal{H}$ respectively.

4.5.4 Computational Gain

Under this separation, the overall time complexity becomes $\mathcal{O}(CKdHW + KNdHW)$ or $\mathcal{O}(dK(N+C)HW)$. Now for computational efficiency, the parameter K should be chosen such that:

$$K < \frac{dCN}{C+N} \tag{21}$$

As described in §4.3, for keeping the functionality of the modified model as close as the original model, the reconstruction error of the convolution filter should be very low, i.e.,

$$||\mathcal{K} - \hat{\mathcal{K}}||_2^2 \approx 0 \tag{22}$$

[3]The $4D$ filter bank $\mathcal{K}$ can be viewed as a collection of N $3D$ filters.

Platform	RAM	CPU		GPU	
		Cores	Speed	Cores	Speed
Snapdragon	1 GB	4	1.2 GHz	6	450 MHz
Tegra	1 GB	4	2.3 GHz	192	950 MHz
Cortex M0	8 KB	1	48 MHz	–	–
Cortex M3	32 KB	1	96 MHz	–	–

Table 1: Summary of the Hardware Platforms

4.5.5 Runtime Adaptation

As part of SparseSep, we developed a runtime framework that dynamically selects convolution layers and their separation criteria to adapt the overall computations needed for a CNN according to the current availability of computation resources or constraints. The runtime adaptation approach follows very closely the binary search procedure outlined in Algorithm 1. Instead of the fully connected layers, the algorithm begins by constructing a list of available convolution layers. An upper bound of the filter separation parameter K is estimated using Equation 21. $\mathcal{H}$ and $\mathcal{V}$ filters are estimated for the selected separation parameter K as given in [31]. Next, the current convolution layer is now replaced with two successive convolution layers with filters $\mathcal{V}$ and $\mathcal{H}$ and the updated model performance is computed on the validation dataset. If the error criterion is satisfied, the modification is accepted and the procedure moves on to the next convolution layer.

Runtime convolution operation optimization opens up new opportunities for SparseSep to gracefully shape and control resource consumption of both DNNs and CNNs on a large variety of hardware platforms, which is not possible for systems like DeepX [14].

5. EVALUATION

In this section we systematically summarize results from a number of experiments to highlight the main benefits of the sparse weight factorization and convolution separation techniques presented above. As the target wearable hardware we consider four platforms: (i) Qualcomm Snapdragon 400 SoCs, (ii) Nvidia Tegra K1, (iii) ARM Cortex M0 and (iv) ARM Cortex M3. Hardware specifications for the four platforms are summarized in Table 1.

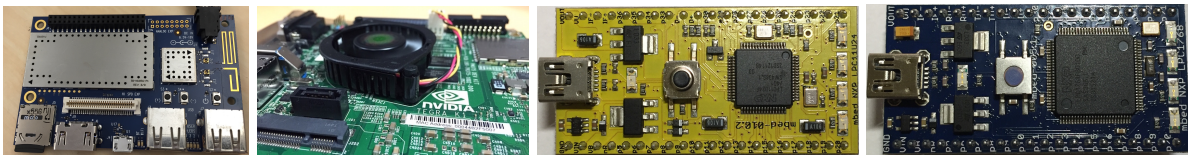
5.1 Experimental Setup

In this paper we focus on audio inference as the representative wearable and mobile sensing tasks to infer the situational context of users' surroundings. We train and deploy DNN models to recognize the ambient environment of the user and to identify the speaker from voice recordings. We apply our factorization technique to CNN models by considering two state-of-the-art object recognition models: AlexNet and VGG. However, do could not run the CNN models on the ARM Cortex platforms for severe memory limitations (allowing only 8 KB and 32 KB).

5.1.1 Audio Datasets

We consider two large publicly available audio datasets: (i) LITIS Rouen Audio scene dataset [32] and (ii) Automatic Speaker Verification Spoofing and Countermeasures Challenge Dataset [33].

The Audio Scene dataset, referred in this paper as the Ambient dataset, contains over 1500 minutes of audio scenes,



(a) Snapdragon 400 **(b)** Tegra K1 **(c)** Cortex M0 **(d)** Cortex M3

Figure 4: Hardware platforms: (a) Qualcomm Snapdragon 400 and (b) Nvidia Tegra K1 are used to evaluate the proposed scaling benefits of weight factorization and convolution separation techniques. Additionally, we use (c) ARM Cortex M0 and (d) M3 to evaluate DNN performances.

which were captured using Samsung Galaxy S3 smartphones. The dataset is composed of 19 different ambient scenes such as, 'plane', 'busy street', 'bus', 'cafe', 'student hall' and 'restaurant'. Audio measurements were recorded with a sampling frequency of 22.05 KHz and several 30 seconds long audio files from each ambient environment are made available.

The Audio Speaker Verification dataset, referred in this paper as the Speaker dataset, contains speech recordings from 106 individuals (45 male and 61 female). In addition to the clean voice recordings, the dataset also contains synthesized data for conducting spoofing attacks, however, in this paper we only focus on clean audio recordings from all 106 participants. Audio measurements were recorded with 16 KHz sampling frequency. To maintain a near equal class distribution, we restrict the maximum duration of audio recording to 15 minutes per user.

For the CNN models, we dowloaded pre-trained models from the caffe zoo repository and we use the original test dataset to measure the overall recognition performances of these models. In Table 2 we summarize all deep models studied in this work.

5.1.2 Deep Architecture Training

Small neural networks, e.g., models with a single hidden layer, can be efficiently trained using the well known back-propagation algorithm [34]. However, deep architectures, especially with many hidden layers, are difficult to train and development of efficient training algorithms are an active area of research in machine learning. Past years have seen the development of unsupervised pre-training approaches for better intialization of the layer weights. In this work we use *denoising autoencoders* to pre-train the weights of the deep architecture and apply the back-propagation algorithm to fine tune architecture for classification purposes. Before training the audio models, we follow a sliding window approach, as described in [32], to extract 13 *mel-frequency cepstral coefficients* (MFCC) from a measurement window of 25 milli seconds. The extracted MFCC features are then aggregated over a 5 second period to generate an input feature dimension of 650, see [32] for details. For both the dataset we follow the same data pre-processing approach. Finally, we train DNN models with two hidden layers (each having $1,000$ nodes) on both datasets and use early stopping criteria to avoid over-fitting.

5.1.3 Hardware

We evaluate the performances of the weight factorization and convolution separation techniques, while executing DNNs and CNNs, on Qualcomm Snapdragon 400 [35] and Nvidia Tegra [15]. To highlight the benefits of sparse factorization,

Name	Type	Parameters	Architecture
AlexNet	CNN	60.9M	c:5$^{\imath}$; p:3‡; fc:3*
VGG	CNN	138.4M	c:13$^{\imath}$; p:5‡; fc:3*
Ambient	DNN	1.7M	fc:3*
Speaker	DNN	1.8M	fc:3*

$^{\imath}$*convolution layers*; ‡*pooling layers*; **fully connected layers*

Table 2: Representative Deep Models

we also perform experiments by running the DNN models on ARM Cortex M0 and M3.

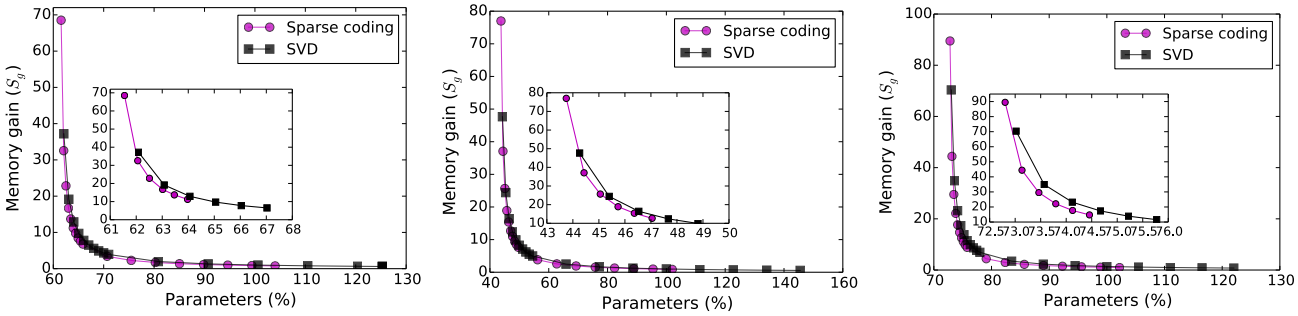
The Snapdragon 400 SoC [35] is widely available in many smartwaches, e.g., LG G smartwatch R [36]. Figure 4a shows a snapshot of the Snapdragon development board used in our evaluations. Primarily designed for phones and tablets, it contains 3 processors: a Krait 4-core 1.2 GHz CPU, an Adreno 306 GPU and a 680 MHz Hexagon DSP. We find the CPU can address 1GB of RAM, but the DSP only 8MB. All our experiments on Snapdragon were conducted using its CPU only.

Although, not as popular as the Snapdragon, the Tegra K1 [15] (Figure 4b) provides extreme GPU performance, unseen in other mobile SoCs. The heart of this chip is the Kepler 192-core GPU, which is coupled with a 2.3 GHz 4-core Cortex CPU and an extra low-power 5th core (LPC). The K1 SoC is used in the Nexus 9, Google's phone prototype within Project Ara [37], and even high-end cars [38]. It is also used in IoT devices like the June Oven [39]. Executing code on the LPC requires the toggling of linux system calls, while access to the GPU is available from CUDA drivers [40].

The ARM Cortex-M series are examples of ultra-low power wearable platforms. The smallest of them all is Cortex M0, which consumes 12.5 μW/MHz and support a memory size of 8 KB (Figure 4c). The M3 variant (Figure 4d) of the cortex has double processing abilities (96 MHz) and support a 32 KB memory. These low-end micro-controllers often have limited memory management capabilities. In our experiments we could only use around 5.2 KB memory on Cortex M0 and around 28 KB memory on Cortex M3. Availability of a small memory requires frequent paging, while executing a large model. The I/O capabilities of Cortex M0 was found to be significantly slower than Cortex M3. For prototyping, we use MBED LPC11U24 [41] and LPC1768 [42] boards for running experiments on Cortex M0 and M3 respectively.

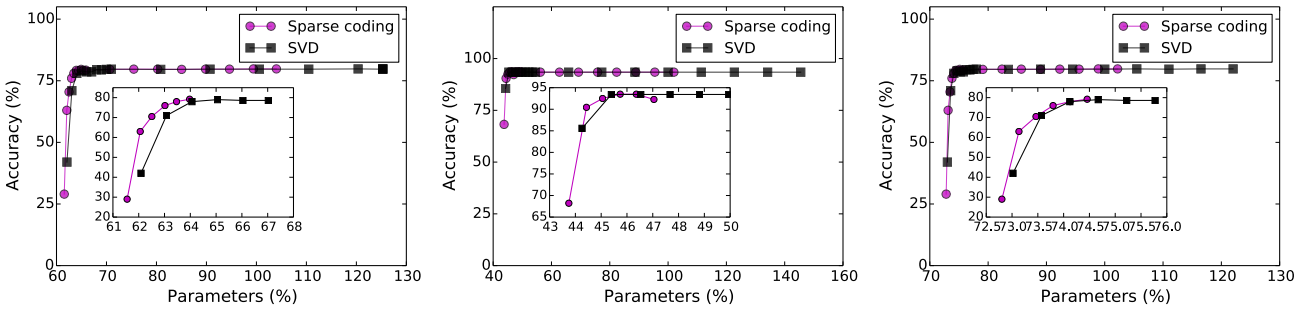
5.2 Scalability Under Sparse Factorization

Next, we study the accuracy of modified DNN model and its space requirements under factorization of fully connected layer weights. Although, the principle of weight factorization can be simultaneously applied to all fully connected



(a) Ambient (DNN) **(b)** Speaker (DNN) **(c)** AlexNet (CNN)

Figure 5: Comparison of memory gains with reduced hidden layer nodes when using sparse coding- and SVD-based weight factorizations for DNNs and CNNs. For simplicity of illustrations, factorization of one fully connected layer of all three models are considered. For all the models, sparse factorization generated much smaller model compared to the SVD-based factorization.



(a) Ambient (DNN) **(b)** Speaker (DNN) **(c)** AlexNet (CNN)

Figure 6: Comparisons of recognition accuracy performances with reduced hidden layer nodes when using sparse coding- and SVD-based weight factorizations for DNNs and CNNs. In our experiments we allow a maximum of 5% degradation in accuracy from the original model performance. Sparse factorization maintains high accuracy, similar to SVD approach, but generates models with smaller memory footprint. Similarly as above, we factorize only one layer.

layers of a DNN, for simplicity of understanding, we only focus on one layer of the DNN and study the effect of the factorization.

Figure 5 illustrates the space or memory gain that can be achieved over the original models (two DNNs and one CNN) under sparse and SVD-based factorizations for various sizes of the new inserted hidden layer. Note that, the high range in Y-axis in both the figures indicate that the sparse coding-based approach provides a significantly better scalability over the SVD solution (see the insets). For example, the smaller the number of nodes kept in the inserted hidden layer, higher is the gain in memory. Additionally, the gain also comes from the sparse matrix multiplications and in this experiment we keep the sparsity level at 20%, while performing the dictionary-based weight factorization. Sparse coding is seen to consistently maintain a superior gain over SVD. Figure 5 also indicates that a larger value of nodes in the inserted hidden layer can adversely increase the space requirement of the model, thus making it inefficient. Finally, the criterion for selecting a good value for k, as given in Equation 7, can also be empirically understood from the figures. SVD looses memory gain when k retains around 80% of the hidden layer parameters. For a layer with $1,000$ nodes, a k value of 80% will results in less than 400 nodes in the inserted new layer. From Equation 7 we empirically find $k = \frac{m \cdot n}{m+n} = \frac{650*1000}{650+1000} \approx 394$, thus verifying the mathematical intuition given earlier.

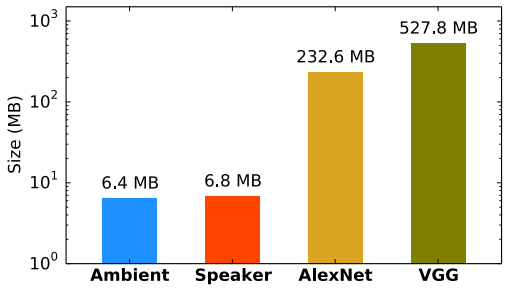
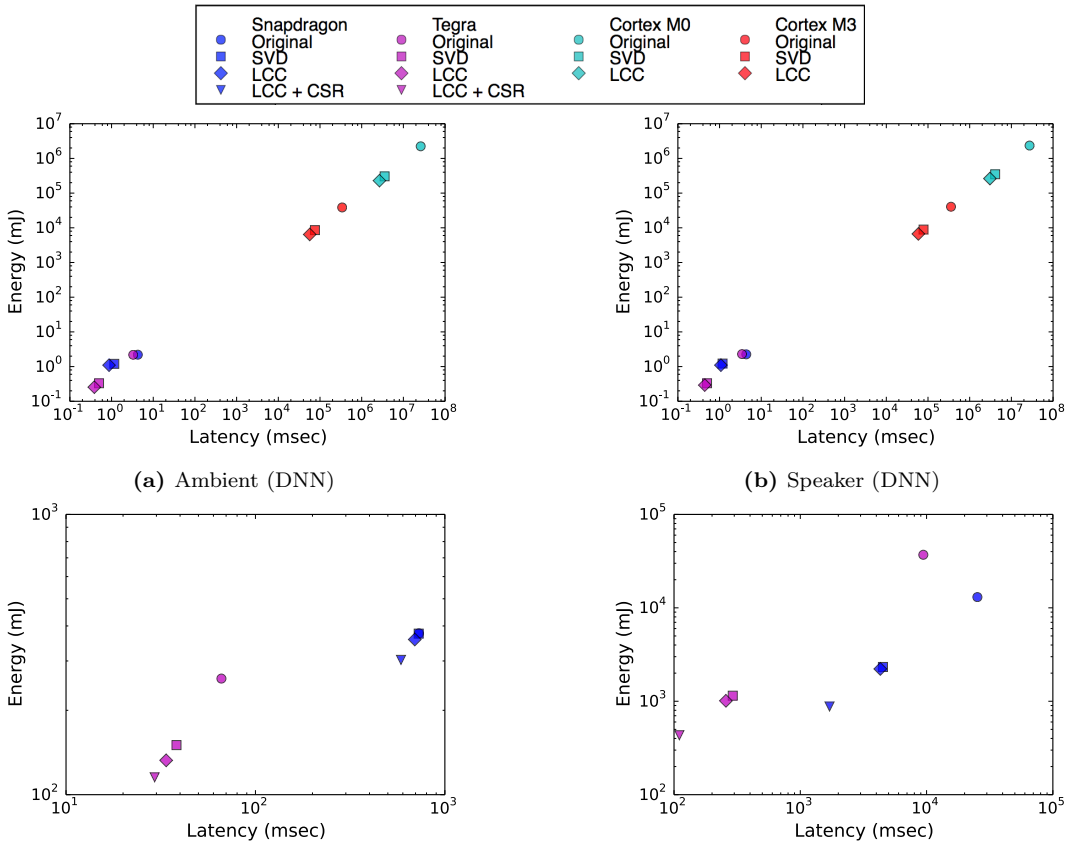


Figure 7: Memory requirements of four original deep learning models studied in the paper. Ambient and Speaker are two audio-based DNN models, whereas AgeNet and AlexNet are two image-based CNN models.

Seeing sparse factorization giving rise to a great memory benefits, we next study the effect of factorization on model performance. Figure 6 illustrates the accuracy performance of the models (same two DNNs and one CNN as given in Figure 5) under both types of weight factorizations. As indicated before, under faithful factorization of the model, i.e., $\boldsymbol{W}^L = \boldsymbol{U} \cdot \boldsymbol{V}$, the modified model would have the same functional properties, i.e., same accuracy. However, as we search for small k the reconstructions starts to deviate significantly from $\boldsymbol{W}^L$ and the model accuracy is observed to violate the



(a) Ambient (DNN)

(b) Speaker (DNN)

(c) AlexNet (CNN)

(d) VGG (CNN)

Figure 8: Energy-consumption and average inference time of different variants of the deep models on four different hardware platforms.

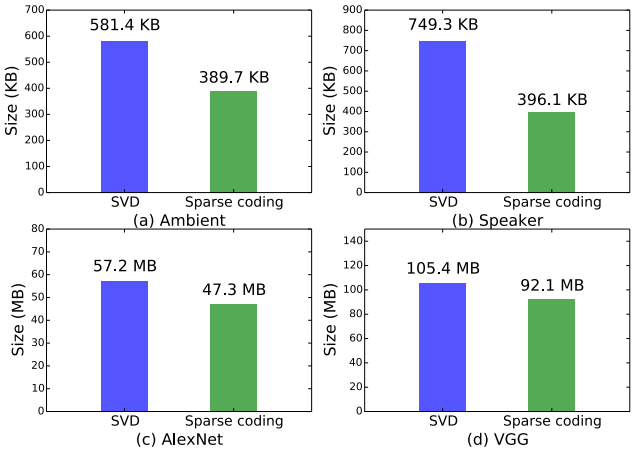


Figure 9: Memory requirements of the four deep models under SVD- and sparse weight factorizations

accepted (pre-defined) 5% tolerance level. Interestingly, for majority of the chosen values of k, both sparse coding and SVD exhibit accuracy very close to the original DNN. Thus, the factorization approach can reduce memory footprint significantly, while maintaining high accuracies.

Figure 7 illustrates the original (uncompressed) model sizes for all the four DNN/CNN models considered in this work. The two DNN models are much smaller in size and depth, compared to the CNN models. However, after our factorizations the memory footprints of the models are highlighted in Figure 9. To obtain an optimized model we execute Algorithm 1 with the models and allow an accuracy tolerance level of 5%, the algorithm performs factorization on all fully connected layers of the model and generates a compact version of the model. This figure indicates the ability of the Sparse factorization to achieve significantly smaller and equal functional models as provided by the SVD technique. The memory is one of the main bottlenecks in low end platforms such as Cortex M series, and thus Sparse coding technique allows for a better tradeoff solutions by significantly reducing the amount of required paging.

Although, we focussed mainly on the factorizations of DNN layer weights, the basic idea applies to a subset of layers within CNN models. For example, in addition to the convolution and pooling layers, often a CNN has one or more fully connected layers, e.g., for classification purposes. These fully connected layers are ideal for sparse factorizations to gain computational and memory efficiencies. Thus, as the final set of experiments we apply our proposed sparse factorization on the AlexNet model, which is a popular computer vision (CNN) model trained for recognizing objects within natural images. However, later we show that the main computational benefits of CNN models arise mainly from the separation of kernel approach.

Contrary to the model used for Ambient or Speaker identification, the original AlexNet model is much larger, e.g., contains 61 million parameters [12], and requires a storage space of 233 MB. When applied sparse factorization to its first fully connected layer only, memory requirement reduces from 233 MB to below 100 MB. Figure 5c and 6c respectively shows memory gain and accuracy deviations for various sizes of the inserted hidden layer.

5.3 Runtime and Energy Performances

We now turn our attention to evaluating runtime and energy performances of sparsely factorized DNNs and CNNs on the four wearable hardware platforms (Cortex M series are only used to evaluate DNNs). For the CNN models we also evaluate the performance of the convolution separation approach on runtime and energy consumption.

To measure runtime and energy performances, we next run the original, the factorized models, and the convolution separated models (incase of CNNs) on the four hardware platforms. The results of the experiment are presented as a tradeoff study in Figure 8 for all four individual models. Not only space scalability, the reduced number of parameters significantly improves the running time of the sparse model on all platforms (note the log scale on both the axes). For both the ambient and speaker models (DNNs), the average[4] inference time is observed to vary significantly across platforms. However, the effects of factorizations of DNNs are significant on all platforms. Overall, the sparse factorization generating better running time over SVD. On both Snapdragon and Tegra, the factorized DNNs runs under one milli second, resulting in around 5 times faster inference than the unmodified model. Similarly to the running time, sparse factorization helps to drop the average energy consumptions significantly on all platforms. For example, on Cortex M0, the power consumption becomes one tength. On Snapdragon 400, the optimized DNN model now consumes only 56% of energy compared to the unmodified model.

Most interesting performance gains are observed for the CNN models, while applying the convolution separation (CSR) technique in conjunction with the weight factorization (LCC). On both Snapdragon and Tegra, the VGG model is seen to be benefitted the most, as it contains significantly high (13) numbers of convolutional layers than AlexNet (see Table 2). The overall running time of the optimized VGG model is just over 1.5 sec on Snapdragon, which is around 2.7 times faster (little below the theoretical upper limit of 3) than its sparse weight factorized only variant.

Thus the techniques presented in the paper open up new opportunities to drastically reduce the memory footprint and overall computational demands of state-of-the-art deep models. We believe that our work will make deep learning based inference engine on wearable and IoT devices highly popular and will help to redefine mobile and IoT experiences.

6. DISCUSSION

We briefly examine a range of key issues related to SparseSep, along with the limitations of our approach.

Broader Deep Learning Support. Our approach has been tested on the two most popular forms of deep learning – DNNs and CNNs – which both include fully-connected feed-forward layers. Any deep model that includes this layer type will benefit from sparse weight factorization of SparseSep. Moreover, the convolution separation technique of SparseSep will allow further optimization of CNNs. Although the mixture of layer types within a model will influence the gains, for example, CNNs have more convolutional layers than feed-forward layers, thus are benefited the most from SparseSep; however, feed-forward layers in CNNs account typically for 80 to 90% of all the memory consumed [12, 20] making them an important target for memory-centric optimizations. Extending the ideas of SparseSep to other forms of deep learning, such as RNNs and LSTMs, remain as important future work.

Hardware Portability. To keep the usage of SparseSep simple for developers we allow them to express constraints to LCC in terms of accuracy, memory and execution time. However, supporting execution time requires SparseSep to estimate the performance of a specific deep model architecture (i.e., layers and node configuration) on the target hardware. SparseSep includes only a fairly simple estimation process based on data from hardware profile process. In our experiments, profiling hardware takes only around 30 minutes using an automated script that tests various model architectures while the device is attached to a power monitor.

Hardware Accelerators. Purpose-built hardware accelerators for deep learning are beginning to emerge [43, 6]. While none of them are currently suitable for wearables yet; more importantly, SparseSep will remain useful even when accelerators become available due to the substantial savings in resource usage offered with only a minimal impact on accuracy. This will allow accelerators to execute even larger models than currently possible. Moreover, the reductions in resources enabled by SparseSep will facilitate the design of more energy efficient deep learning accelerators, better suited to wearables than today, because they can be built with fewer computational and memory resources than previously possible.

7. RELATED WORK

We now overview work closely related to SparseSep, this includes results from the compressive sensing area, and efforts to lower resources used by sensing algorithms.

Optimizing Mobile Sensing Algorithms. The challenge to mobile resources of executing the algorithms necessary to extract context and user activities from sensor data has been long recognized and studied. Approaches include the development of sensing algorithm optimizations such as short-circuiting sequences of processing or identifying efficient sampling rates and duty cycles for sensors and algorithm components like [44, 45, 46]. Work such as [47] aims to combine such optimizations along with careful usage of energy efficient hardware. Others [48] take a stream view and so applies stream oriented optimizations on the basis of real dataflows arriving from sensors. [49] approaches performance tuning by extending ideas of feature and model selection to consider device resources, and even cloud offloading opportunities.

In sum, SparseSep is the most recent of this chain of work, but differs importantly in the fact that it is one of the few

[4] All inferences are repeated 1,000 times and we report their averages in the figures.

to investigate deep learning specific methods exclusively – this in turn allows SparseSep to highlight significant opportunities for gains (such as memory and computation) that only exist within such models. It is likely that many of the other techniques described could operate in combination with SparseSep given they frequently treat the learning algorithm itself as a black-box. However, they may bring undesirable negative side-effects such as higher levels of accuracy loss. More broadly, one could characterize the de-facto approach to using deep learning in wearables and mobile devices today as being based on cloud-based, and therefore approaches like MAUI [50] and those related to it are applicable. However, such approaches suffer from problems including challenges to privacy protection when partitioning deep models as highlighted in §1.

Deep Learning under Resource Constraints. Examples of deep models designed specifically for wearable constraints are still maturing. A popular approach is for experts to hand optimize specific deep models targeting a specific device class (e.g., smartphone). This has been done for machine translation [10], speaker identification [51] and keyword spotting [11] (i.e., a device constantly waiting for a small number of specific phrases). Moderately-sized DNNs designed specifically for constrained processors (like the DSP) in smartphones have also be demonstrated [8, 52] for a range of audio sensing tasks, as well as basic activity recognition scenarios. Discussions of the techniques used to optimize these models are slowly now being reported [20]. In contrast, SparseSep enables *automated* optimization broadly applicable to deep model families (DNN and CNN). There is little cost for SparseSep to be applied to new/revised models, or when resources change (e.g., a new device is released).

Naturally, work is also increasing in the area of more systematic easily re-used techniques for lowering resource consumption in deep models. A common framework for these approaches is from the machine learning community and called *model compression* that most often used at training time (unlike SparseSep that is inference-time focused) to scale to larger datasets or increase hardware utilization. Techniques of this type vary in particular to the extent to which the underlying model is altered. For example, [24] actually removes nodes and reshapes layers within the model while [21, 53] performs types of quantization of parameters within layers. We design SparseSep towards minimizing the modifications made to the model and so adopt approaches that *insert* new layers designed to optimize performance. SVD-based methods of this type are the current state of the art (such as [54]) which SparseSep has numerous advantages as detailed throughout this paper. [14] focuses on how to partition deep learning models across different types of processors (GPUs, CPUs, DSPs) found within an system-on-a-chip; it uses an SVD-based model compression approach to allow a partition of a model to fit within the resources of a specific processor (such as a DSP). SparseSep in contrast is agnostic to processor type, and complementary to [14] in that its performance could only improve if it incorporated ideas of utilizing a spectrum of processors.

8. CONCLUSION

In this paper, we have proposed SparseSep – a set of novel techniques for optimizing large-scale deep learning models – that allows deep models to function even under the extreme system constraints presented by wearable hardware. Conceptually, SparseSep brings many of the recent advances in sparse dictionary learning and convolution kernel separation to how deep learning models are both represented and executed. The core innovation is an off-line method to sparsify the internal feed-forward layers of both DNNs and CNNs, and optimization of convolution filters of CNNs, that produces a highly compact model representation with only small reductions in classification accuracy. Such layer sparsity in turn, enables SparseSep to re-invent the inference process used in deep models and allow classification to occur with radically lower resources. We believe the leaps in deep learning inference efficiency that SparseSep provides will prove to be a significant enabler for the broader adoption of such modeling techniques within mobile and IoT platforms.

9. ACKNOWLEDGEMENTS

We would like to thank our Shepherd, Prof. Bhaskar Krishnamachari, for his valuable comments and time to improve the overall quality of this paper. We also thank Ray Kearney from Nokia Bell labs, Dublin for helping us in building the power monitoring tool for ARM Cortex boards.

10. REFERENCES

[1] Y. Bengio, I. J. Goodfellow, and A. Courville, "Deep learning," 2015, book in preparation for MIT Press. [Online]. Available: http://www.iro.umontreal.ca/~bengioy/dlbook

[2] 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

[3] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[4] 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.

[5] K. J. Geras, A. Mohamed, R. Caruana, G. Urban, S. Wang, Ö. Aslan, M. Philipose, M. Richardson, and C. A. Sutton, "Compressing lstms into cnns," *CoRR*, vol. abs/1511.06433, 2015. [Online]. Available: http://arxiv.org/abs/1511.06433

[6] 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

[7] N. Hammerla, J. Fisher, P. Andras, L. Rochester, R. Walker, and T. Plötz, "Pd disease state assessment in naturalistic environments using deep learning," in *AAAI 2015*, 2015.

[8] N. D. Lane and P. Georgiev, "Can deep learning revolutionize mobile sensing?" in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '15. New York, NY, USA: ACM, 2015, pp. 117–122. [Online]. Available: http://doi.acm.org/10.1145/2699343.2699349

[9] "Your Samsung SmartTV Is Spying on You, Basically," http://www.thedailybeast.com/articles/2015/02/05/your-samsung-smarttv-is-spying-on-you-basically.html.

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

[11] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, ser. ICASSP'14, 2014.

[12] 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

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2012, pp. 1223–1231.

[14] 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*.

[15] "Nvidia Tegra K1," http://www.nvidia.com/object/tegra-k1-processor.html.

[16] "Arm Cortex-M3," http://www.arm.com/products/processors/cortex-m/cortex-m3.php.

[17] B. A. Olshausen and D. J. Field, "Sparse coding with an overcomplete basis set: A strategy employed by v1?" *Vision research*, vol. 37, no. 23, pp. 3311–3325, 1997.

[18] *Principal component analysis*. Wiley Online Library, 2002.

[19] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2007.

[20] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[21] J. S. Ren and L. Xu, "On vectorization of deep convolutional neural networks for vision tasks," in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT press, 2009.

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

[24] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu, "Reshaping deep neural network for fast decoding by node-pruning," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 245–249.

[25] H. Lee, A. Battle, R. Raina, and A. Y. Ng, "Efficient sparse coding algorithms," in *Neural Information Processing Systems (NIPS)*, 2007.

[26] R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng, "Self-taught learning: Transfer learning from unlabeled data," in *Proceeding of the International Conference on Machine Learning (ICML)*, 2007.

[27] S. Bhattacharya, P. Nurmi, N. Hammerla, and T. Plötz, "Using unlabeled data in a sparse-coding framework for human activity recognition," *Pervasive and Mobile Computing*, May 2014.

[28] M. Aharon, M. Elad, and A. Bruckstein, "K-svd: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311–4322, 2006.

[29] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, "Learning separable filters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013, pp. 2754–2761.

[30] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[31] C. Tai, T. Xiao, Y. Zhang, X. Wang, and W. E, "Convolutional neural networks with low-rank regularization," *arXiv preprint arXiv:1511.06067*, 2015.

[32] A. Rakotomamonjy and G. Gasso, "Histogram of gradients of time-frequency representations for audio scene detection," Technical report, HAL, https://sites.google.com/site/alainrakotomamonjy/home/audio-scene, 2014.

[33] T. E. N. Y. J. Wu, Zhizheng; Kinnunen, "Automatic speaker verification spoofing and countermeasures challenge (asvspoof 2015) database," University of Edinburgh. The Centre for Speech Technology Research (CSTR), Tech. Rep., 2015.

[34] R. J. W. David E Rumelhart, Geoffrey E Hinton, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[35] "Qualcomm Snapdragon 400," https://www.qualcomm.com/products/snapdragon/processors/400.

[36] "LG G Watch R," https://www.qualcomm.com/products/snapdragon/wearables/lg-g-watch-r.

[37] "Google Project Ara," http://www.projectara.com.

[38] "Audi self-driving car brings NVIDIA Tegra K1 front and center," http://www.slashgear.com/audi-self-driving-car-brings-nvidia-tegra-k1-front-and-center-25322090/.

[39] "June Oven," http://techgage.com/news/nvidias-tegra-k1-soc-has-made-it-into-an-oven-that-detects-what-its-cooking/.

[40] "Nvida CUDA," http://developer.nvidia.com/cuda-zone.

[41] "ARM MBED Cortex M0," https://developer.mbed.org/platforms/mbed-LPC11U24/.

[42] "ARM MBED Cortex M3," https://developer.mbed.org/platforms/mbed-LPC1768/.

[43] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[44] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/1378600.1378630

[45] S. Nath, "Ace: Exploiting correlation for energy-efficient and continuous context sensing," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307640

[46] M.-R. Ra, A. Sheth, L. B. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices." in *MobiSys*, A. K. Agrawala, M. D. Corner, and D. Wetherall, Eds. ACM, 2011, pp. 43–56. [Online]. Available: http://dblp.uni-trier.de/db/conf/mobisys/mobisys2011.html#RaSMPWG11

[47] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing, "Orbit: A smartphone-based platform for data-intensive embedded sensing applications," in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, ser. IPSN '15. New York, NY, USA: ACM, 2015, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/2737095.2737098

[48] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song, "Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '12. New York, NY, USA: ACM, 2012, pp. 211–224. [Online]. Available: http://doi.acm.org/10.1145/2426656.2426678

[49] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao, "Balancing energy, latency and accuracy for mobile sensor data classification," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '11. New York, NY, USA: ACM, 2011, pp. 54–67. [Online]. Available: http://doi.acm.org/10.1145/2070942.2070949

[50] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814441

[51] E. Variani, X. Lei, E. McDermott, I. L. Moreno, and J. Gonzalez-Dominguez, "Deep neural networks for small footprint text-dependent speaker verification," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, Florence, Italy, May 4-9, 2014*, 2014, pp. 4052–4056. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2014.6854363

[52] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: Robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. New York, NY, USA: ACM, 2015, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/2750858.2804262

[53] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[54] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition," in *Interspeech*, 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=201364