DXTK: Enabling Resource-efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit

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

ABSTRACT

Deep learning is having a transformative effect on how sensor data are processed and interpreted. As a result, it is becoming increasingly feasible to build sensor-based computational models that are much more robust to real-world noise and complexity than previously possible. It is paramount that these innovations reach mobile and embedded devices that often rely on understanding and reacting to sensor data. However, deep models conventionally demand a level of system resources (e.g., memory and computation) that makes them problematic to run directly on constrained devices.

In this work, we present the DeepX toolkit (DXTK); an open-source collection of software components for simplifying the execution of deep models on resource-sensitive platforms. DXTK contains a number of pre-trained low-resource deep models that users can quickly adopt and integrate for their particular application needs. It also offers a range of runtime options for executing deep models on range of devices including both Android and Linux variants. But the heart of DXTK is a series of optimization techniques (viz. weight/sparse factorization, convolution separation, precision scaling, and parameter cleaning). Each technique offers a complementary approach to shaping system resource requirements, and is compatible with deep and convolutional neural networks. We hope that DXTK proves to be a valuable resource for the community, and accelerates the adoption and study of resource-constrained deep learning.

CCS Concepts

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

Keywords

Wearables; Mobile Sensing; Deep Learning; Toolkit

1. INTRODUCTION

Increasingly, the state-of-the-art in the modeling of noisy complex data is shifting to deep learning [11, 19] principles and techniques. Through a series of innovative deep neural network architectures and algorithms, domains such as object and face recognition [46, 31], machine translation [20], and speech recognition [27] have been transformed – with the resulting new models demonstrating significant improvements in accuracy and robustness. However, adoption of these methods within mobile and embedded systems has lagged notably; even though many of these systems require precise sensor inference capabilities. The core reason for this is the intrinsic complexity of deep models and the heavy demands on computational and memory resources that this complexity entails. Deep Neural Networks [28] (DNNs) and Convolutional Neural Networks [36] (CNNs), for example, routinely are composed of thousands of interconnected units, and millions of parameters [31, 46]. Consequently, most wearable, mobile and embedded sensor-based systems today adopt simpler modeling approaches (such as Decision Trees and Gaussian Mixture Models [14]) that have a lower resource footprint – even though they are often known to have inferior performance relative to deep learning alternatives.

In this paper, we present the design, implementation and evaluation of the DeepX toolkit (DXTK) [2]; a collection of open-source software components that aim to ease the adoption and study of deep learning algorithms within mobile, wearable and embedded platforms – particularly with respect to inference-time model performance. The core of DXTK is a series of Model Optimizers each of which implement various techniques towards reducing the memory and computational demands of a deep model – typically at the expense of a small loss in accuracy (tunable by the user). In this initial release of DXTK there are five optimizers included, specifically: *weight factorization* [32], *sparse factorization* [12], *convolution separation* [45, 12], *precision scaling* and *parameter cleaning*; the set of Model Optimizers within DXTK is designed to be extensible so that new innovations can be incorporated as they are developed. Similarly, DXTK is also compatible with existing deep learning frameworks that are used, for example, to train models with large-scale data using sophisticated deep methods in a relatively user-friendly manner. Currently, DXTK integrates directly with Torch [8] and adopts the Torch model format. General support for various deep learning frameworks is an underlying design assumption and TensorFlow [10] support is expected shortly.

Within DXTK, Model Optimizers are accompanied by a collection of pre-trained pre-optimized Low-resource Models. These offer users deep models capable of many common sensing tasks, such as speaker identification or object recognition, that are based on well-known and state-of-the-art techniques but have already been optimized through the careful application of DXTK techniques. Users of DXTK can simply integrate these models into their applications if they wish to leverage deep learning technology as quickly as possible. Runtime Support is the final DXTK software component; two primary forms of support are offered. First, DXTK enables Torch to operate as a prototyping-friendly inference engine under both Android (for phone and watch support) and Linux-based systems (ideal for Internet-of-Things development). Second, for those platforms where the overhead of Torch is unacceptable, DXTK includes a thin inference-only C++ runtime. Although only DNNs are supported as part of this initial release.

MobiCASE 2016, November 30-December 01, Cambridge, Great Britain
Copyright © 2016 EAI
DOI 10.4108/eai.30-11-2016.2267463

2. BACKGROUND

We begin with a primer on deep learning methods, before describing challenges in implementing them on constrained devices.

Deep Neural Networks. An architecture of a deep model includes a series of (often, but not always) fully-connected layers, and within each layer comprising some number of units (or nodes). Each unit has a state (an activation) that is dictated by all units in the prior layers. An example of such a deep model is shown in Figure 1. The input layer of units (the first layer in the architecture) are set by raw (or gently normalized) data – for instance, an audio clip to be classified as a particular type of sound. The output layer of the model (the final layer of the architecture) determines the category of the data (assuming a discriminative task is being performed). Such data categories often correspond to individual units of the output layer. In-between the input and output layer are what are refereed to as hidden layers. The role of these layers are to convert the data initialized input layer into the final activation of a unit within the output layer (that indicates the category). Adjacent layers within a DNN are connected via a collection of parameters (specifically, weights and biases). The activation of one unit is determined by these parameters and the state of prior layer units; more precisely, for successive fully connected layers with i and j units there are i X j weight parameters and k bias parameters. As a result, the total number of parameters for a typical DNN containing tens of layers and hundreds of units per layer can easily reach into the millions.

Using a DNN to classify a segment of data (i.e., perform inference) is done with the feed-forward algorithm. For each segment of data (be it an audio frame or image) the algorithm is performed repeatedly, and in isolation. Feed-forward begins at the input layer and updates the state of each layer iteratively based on the prior one, depending on the parameter values. The activation state of each unit within a layer is updated successively until the final unit within the final layer is reached. At this point the inference class is usually the output layer unit with the largest activation.

Convolutional Neural Networks. An equally popular, yet alternative form, of deep learning is a CNN – an example of which is shown in Figure 2. Mostly they are used for tasks related to images where this approach offers often state-of-the-art performance [36] – however, the usage of CNNs is increasing into other learning tasks. CNNs contain one or more of the following collection of layer types: convolutional layers, pooling or sub-sampling layers, and fully connected layers. Note this final layer type (fully-connected) are identical to those described above in relation to DNNs). At the core, CNNs aim to extract a series of representations (i.e., learned features) from input 2D images. The architecture seeks to convert images to representations of increasingly higher abstraction towards target learned concepts (such as discriminating between objects). Accomplishing this is done by initially applying a number of convolutional filters to an input image to extract local data properties. Subsequently, min and max pooling is usually applied to extract summary characteristics from the convolution representations. This results in the extracted features being invariant to, for example, translation and undergoing a type of dimensionality reduction. Sigmoid non-linearities are often applied as well as biases added prior to pooling being done. The number of parameters for a CNN are based on the count and dimension of convolutional filters, as well as how they are applied to data. Nevertheless, just as in the case of DNNs the parameter count also can easily run into the millions.

In fashion very similar to that of DNNs, inference under a CNN is performed on a single data segment per iteration. Usually input sensor data are vectorized into a 2D representation (which happens

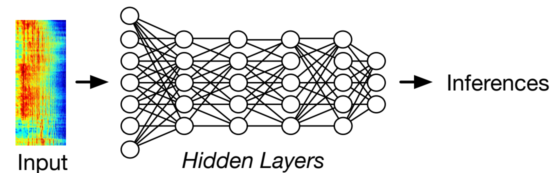

Figure 1: Deep Neural Network Architecture

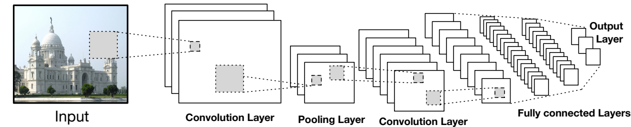

Figure 2: Convolutional Neural Network Architecture

to be highly natural for 2D images). Data are feed into convolutional layers at the start of the architecture. These layers provide a type of pre-proccessing to the data and operate and apply a sequence of filters (or patches), the output of which are fed to fully connected layers at the end of the CNN architecture. Inference is completed in roughly the same manner previously as detailed.

Challenges on Embedded and Mobile Devices. Commonly used deep models, such as Inception [44] used in computer vision tasks like object detection, can take days or even weeks to train on powerful GPU-based desktop machines. For much more constrained devices like smartphones and embedded platforms, even thought they only usually need to perform classification with these models, due to the shear number of layers, units and parameters significant system resource bottlenecks still surface. There is still much to be learned about these bottlenecks, but early studies into these issues [25, 33] observe three key (perhaps unsurprising) forms:

- *Memory:* The most obvious bottleneck for deep learning is memory. The models themselves commonly run in the order of hundreds of MBs due to their complex representation. From a practical point of view, this inflates sensor apps dramatically and normally dwarfs application logic. Users are resistant towards apps that take so long to install and update. If we turn to runtime requirements, peak memory needs often will preclude a particular platforms from using a model even if they have the necessary program storage to hold the model representation. It is not uncommon for a single model layer to require tens or hundreds of MBs and this might be a 90$\times$ more than the average layer consumption within the model. In such cases, manual customization of the model is necessary before it can execute completely to the final layer.

- *Computation:* Models like VGG, even when carefully implemented on embedded platforms, have been seen to take minutes to complete for a single image due to the paging required – although in most cases this is a secondary effect of a large memory footprint. Similarly, smartphones despite having significant computational power present challenges in executing deep models with latency suitable for user interaction and usually realized with hand-optimized models that offload computation to the cloud, or at least a GPU. Computation requirements are also sharply shaped by model architecture, with some layers like convolutional ones being much more compute bound than feed-forward layers (that are more memory-bound). As a result, overlooked changes in architecture can have a significant runtime impact.

- *Energy:* Extending largely from computational concerns, the power-consumption of deep models due to excessive execution time for even a single inference can prohibit them from being used for continuous monitoring fashion although this is often needed. This better (but more complex) deep models to be hidden within sensing system processing chains, only activated when a less resource intensive simpler model (perhaps supporting a narrower range of categories) detects a particular certain condition.

It should be noted, this view of deep learning resource issues is limited by being largely inference-centric. Looking ahead because of the need to adapt models (for example, to recognize the face of a user's child in their personal photos), bottlenecks specific to types of deep model training on constrained devices are likely to also emerge. This will bring even an wider more extreme set of challenges into focus than those described above.

3. DXTK OVERVIEW

The objective of the DeepX toolkit (DXTK) [2] is two-fold. First, it seeks to facilitate the prototyping of mobile and embedded applications that use deep models. For this reason it includes, for example, pre-built deep models suitable for constrained devices that perform typical sensor processing tasks like object recognition. These models can then be easily executed in one of the runtime support options offered, such as within an Android development environment. Second, DXTK supports the investigation of how, in particular, inference-time resource consumption of deep learning models and algorithms can be reduced. Helpful in this regard are the a series of deep model optimization techniques it includes; each offers different approaches for reducing resource usage that can be extended, or benchmarked against freshly proposed alternatives.

3.1 Architecture

As illustrated in Figure 3, DXTK is comprised by three types of software components (viz. Low-resource Models, Model Optimizers, Runtime Support). At this stage, all code is written in Python or Lua and is assumed to be used in conjunction with the Torch framework [8]. However, DXTK is designed to be modular in a way that allows for future compatibility with alternative deep learning frameworks – TensorFlow [10] compatibility is expected shortly.

Low-resource Models. DXTK includes a collection of pre-trained and pre-resource-optimized models. The initial set of models released operate either images or audio data, and perform classification tasks that include: object recognition, speaker identification and ambient sound class discrimination (e.g., music, conversation). For object recognition two popular model architectures are present, AlexNet [31] and VGG [43]. It is expected these models can be customized to application requirements (e.g., recognizing a type of car, place, clothing) through the typical deep learning process of fine-tuning – simple forms of which DXTK includes within the Torch environment. Audio models are trained by the authors of DXTK using publicly available datasets, later in this paper details of the training and modeling techniques used are given in the Section 5. Multiple versions of each model are offered, each with different levels of resource requirements (such as memory constraints) for use on different platforms. Models are specified within the Torch model format, but conversion to other frameworks are relatively simple. It is anticipated this collection of models will grow over time, and include direct contributions from the community.

Model Optimizers. A total of five different Model Optimizers are included in the initial release of DXTK; namely, *weight factorization* [32], *sparse factorization* [12], *convolution separation* [45, 12], *precision scaling* and *parameter cleaning* – in Section 4, we detail each of these optimizers. Each optimizer has the same aim, to reduce inference-time system resources usage of a deep model – specifically: execution-time memory consumption, the storage/memory footprint of the model itself, and the amount of computation required. Optimizers can be applied to any existing DNN or CNN model, and produce as an output a new version of the model that has been manipulated to reduce its resource needs. Optimizers expect models to be described in a Torch compatible format, and produce models in this same form. In cases where the alterations to the model result in an accuracy loss, each optimizer has parameters that can be tuned (by toolkit users) to control the amount of this loss. It is anticipated users will experimentally tune parameters by observing accuracy and general model robustness on the target data of interest. Similarly, it is expected such tuning will also be done to determine a model that fits within the constraints of the particular target platform. Finally, while each technique is separate and so therefore can be applied in combination with others to a single model, the use of one may limit the effectiveness of others.

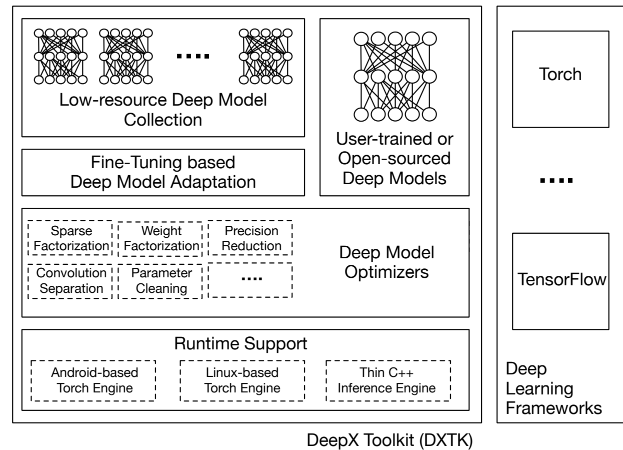

Figure 3: DXTK Architecture.

Runtime Support. The primary runtime support available from DXTK is based on the Torch framework itself. DXTK makes it simple for Torch to be run directly within Android (for smartphones and smartwatch scenarios) and Linux environments (mainly for embedded Internet-of-Things style platforms). It is important to note the support for these environments is based on a mixture of existing open-source code (and in the case of Linux, basic cross-compilation). Therefore, DXTK is acting largely in a role of integrating these opportunities with other DXTK components, and so broadening accessibility to these techniques. Support for Android is based on a customizable template Android application that incorporates Torch for developers. Various template options are offered to change the Torch and Android integration, for example one offering directly a command line Torch prompt useful for testing model variations on end platforms. Although using Torch as an inference engine brings about additional overhead, it also eases the support for a broad range of deep learning forms (especially new innovations) as Torch is maintained by a large existing community. Broadening runtime support is likely to be much simpler under alternative frameworks like TensorFlow because of its native support for platforms like Android, and expanding direct support for processors such as those from Qualcomm, ARM and Intel.

The secondary runtime support present in DXTK is offered for

highly constrained embedded platforms such as the ARM Cortex M0 (see Section 5 for more). This runtime only offers inference execution of DNNs (i.e., feed-forward layers only) and is written in C++ which makes it highly portable. This runtime assumes a Torch model format. Looking ahead we expect to expand the capabilities of this runtime to architectures like CNNs, and develop more sophisticated mangers of memory in particular.

3.2 Usage Models

To further clarify the operation model of DXTK, we now provide brief descriptions of three representative workflows that are anticipated to be popular with users.

Direct Use of Low-resource Models. If one of the Low-resource Models fits the need of the user directly, integration within an system or application under development can be very rapid. The user will need to select the runtime support option that fits the target platform and then iteratively test different versions of the Low-resource Model that are pre-computed for various energy, memory and computational limits. After the model is found, a user can augment the runtime with other application specific logic, for example using language hooks that interact directly with Torch.

Training and Optimizing a Custom Deep Model. Through the use of a conventional deep learning framework, users are able to train a deep model just as they would normally. Although DXTK directly supports Torch model formats, in practice this training can be done by any framework as conversion into the Torch format is fairly simple. Once a user has a candidate deep model to use in a mobile or embedded device they may apply any of the Model Optimizers (offline) to reduce its resource consumption. This will typically be done experimentally, with an optimizer being applied with a certain range of optimizer specific parameters selected. The adapted model produced by the optimizer can then be inspected by the user to determine if the accuracy and resource usage are acceptable. Accuracy can always be assessed on a workstation but certain resources like energy are most easily tested directly on the target device. After model performance is satisfactory the appropriate DXTK runtime can be adopted and customized, with then the model put directly onto the device.

Optimizing an Open-source Deep Model. It is very common for pre-trained deep models and the related code necessary to train them to be released to the public from research and industry groups. In such cases users of DXTK can attempt to integrate these into their system or experiments in a manner very similar to the above workflow. The key distinction is that rather than design and train the model themselves with an existing framework, if a pre-trained model is available they can directly start to apply any DXTK Model Optimizer. (Although if the model format is not compatible with Torch some conversion steps will be required).

4. DXTK MODEL OPTIMIZERS

The technical core of DXTK is a collection deep model manipulation techniques. Each Model Optimizer is based either on previously proposed methods (e.g., [12, 32]) or generally known in the community. We now detail the Optimizers offered in the first release of DXTK.

4.1 Weight Factorization.

Weight factorization (WF) operates on a fully connected layer of DNNs and CNNs. For example, in case of a DNN (see Figure 1) WF can potentially be applied on the connections between any two successive layers. Specifically, all connections between layer L (containing m units) and $L+1$ (containing n units) are represented by the matrix $\boldsymbol{W}^L$, where the ith row of $\boldsymbol{W}^L$ contains all the connection strengths from i^{th} unit in layer L to all units in $L+1$. In case of CNN, WF can be applied on any layers responsible for classification (see Figure 2). WF helps to lower memory and overall computations of deep models by performing *low rank* approximation of the layer weights. As a by product of lesser number of computations, the overall power requirement to execute the deep model also decreases. This technique is a simplification of a method proposed in [32] named Runtime Layer Compression.

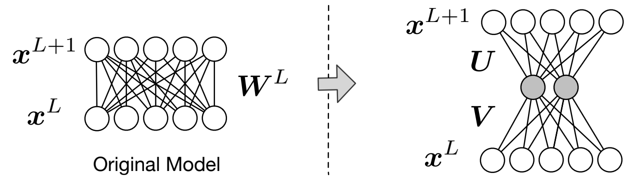

Figure 4: Illustration of the WF technique. Through factorization of the weight matrix, a new layer results and inserted two prior adjacent layers of the original model. The net result of this adaptation is that significantly fewer computations are required at inference time.

Factorization. The key idea in WF is to factorize the weight matrix $\boldsymbol{W}^L$ for layer L into two components, e.g., $\boldsymbol{U}$ and $\boldsymbol{V}$, such that $\boldsymbol{W}^L \approx \boldsymbol{U} \cdot \boldsymbol{V}$, and the overall memory requirement to store the factorized components is smaller than the original weight matrix itself. Additionally, when the states of all units in the layer L is evaluated, the factorized approach needs fewer operations.

We employ the well known *Singular Value Decomposition* (SVD) technique to factorize the weight matrix of a fully connected layer. For instance, the weight matrix $W^{L+1}_{m\times n}$ for two adjacent layers (L and $L+1$) with m and n units respectively (under SVD) can be represented as:

$$W^{L+1}_{m\times n} = U_{m\times m}\, \Sigma_{m\times n}\, N_{n\times n} \tag{1}$$

The weight matrix can be approximated by keeping k biggest singular values, i.e.,:

$$\hat{W}^{L+1}_{m\times n} = U_{m\times k}\, \Sigma_{k\times k}\, N_{k\times n} \tag{2}$$

$$\hat{W}^{L+1}_{m\times n} = U_{m\times k}\, V_{k\times n} \tag{3}$$

Architecture Adaptation. Figure 4 summarizes the weight factorization process, which includes an architectural modification. The weight matrix $W^{L+1}_{m\times n}$ can be replaced by the product of $U_{m\times k}$ and $V_{k\times n}$, which is achieved by introducing a new layer with $k \ll m, n$ units between layer L and $L+1$. Because L and $L+1$ units are fully connected, the introduction of the new layer allows the number of pairwise calculations and weight parameters to fall dramatically, when:

$$k < \frac{m \cdot n}{m+n} \tag{4}$$

Prior empirical [26, 49] and theoretical [30] results support the WF design in two important ways. First, even though the architecture of the model is changed, the impact on downstream layers, trained assuming the original architecture, does not typically result in large increases in error. Second, in fact considerable amounts of compression are possible, if carefully applied to certain layers, before significant accuracy declines are observed. One reason for this is that model representations produced by training processes do not always produce the most compact representation.

4.2 Sparse Factorization

Similarly as WF, sparse factorization (SF) relies on the decomposition of the layer weight matrix, however, requiring one of the component matrices to be sparse[1]. The main benefits of SF over WF are reductions in memory demand and computations. This technique is detailed more completely in [12].

Dictionary Learning. The task of sparse matrix factorization can be formulated as a dictionary learning problem with a sparsity penalty (regularization). Here a dictionary $\boldsymbol{\mathcal{B}}$ is learned from the weight matrix $\boldsymbol{W}^L$ of a fully connected layer using an unsupervised algorithm [13]. Sparse coding approximates an input, e.g., a column of $\boldsymbol{W}^L$, as a sparse linear combination of basis vectors from the dictionary $\boldsymbol{\mathcal{B}}$ (see [12] for details). Thus, $\boldsymbol{W}^L = \boldsymbol{\mathcal{B}} \cdot \boldsymbol{A}$, where $\boldsymbol{A}$ is a sparse matrix.

Architecture Adaptation. Once the dictionary $\boldsymbol{\mathcal{B}}$ and the sparse matrix $\boldsymbol{A}$ is obtained[2], we can follow the same layer adaptation procedure employed by the WF. Figure 4 captures the basic architectural change, although not the level of sparsity that results. Note that sparsity translates into missing connections among units. However, the dictionary size k determines the gain in memory and computations, and the following condition needs to hold to achieve computational and memory benefits:

$$nnz(\boldsymbol{A}) < \frac{n \cdot k}{2}, \tag{5}$$

where, $nnz(\cdot)$ counts the number of non zero elements and n is the number of units in the layer $(L+1)$ [12].

Similarly to WF, the redundancy estimator $\mathcal{E}$ can be used to measure the deviation in estimated weight matrix from the original.

4.3 Convolution Separation

Both WF and SF can reduce the memory requirement of both DNNs and CNNs, however, factorization of fully connected layer does not reduce the bottleneck of massive amount of convolutional computational operations in CNNs. As was the case in the prior technique, this approach was also proposed in [12] – while a closely related technique is also proposed in [45].

Separable Filters. Let $\mathcal{K} \in \mathbb{R}^{N \times d \times d \times C}$ be the set of N convolutional filters (each with dimension $d \times d \times C$) specific to a layer. The output feature map $\boldsymbol{M} \in \mathbb{R}^{N \times H \times W}$ is generated for an input data $\boldsymbol{x} \in \mathbb{R}^{C \times H \times W}$ as:

$$\boldsymbol{M}_j = f\left(\sum_c \boldsymbol{x}^c * \mathcal{K}_j^c + \boldsymbol{b}_j\right), \tag{6}$$

where, $f(\cdot)$ is the non-linear function, $\boldsymbol{b}$ is the bias vector for the layer and c is the index over C input channels. The time complexity for generating $\boldsymbol{M}$ is $\mathcal{O}(CNd^2HW)$.

Here the main goal is to obtain an approximation kernel $\hat{\mathcal{K}}$, i.e., $||\mathcal{K} - \hat{\mathcal{K}}||_2^2 \approx 0$ and $\hat{\mathcal{K}}$ can be decomposed into *horizontal* $\mathcal{H} \in \mathbb{R}^{N \times 1 \times d \times K}$ and *vertical* $\mathcal{V} \in \mathbb{R}^{K \times d \times 1 \times C}$ filters with lower ranks, controlled by the parameter K. Under this condition, the overall convolution can be rewritten as [12]:

$$\mathcal{K}_n * \boldsymbol{x} \approx \hat{\mathcal{K}} * \boldsymbol{x} = \sum_{k=1}^{K} \mathcal{H}_n^k * \left(\sum_{c=1}^{C} \mathcal{V}_k^c * \boldsymbol{x}^c\right) \tag{7}$$

[1] Majority of the matrix elements are 0.

[2] We use the K-SVD algorithm to learn the dictionary. Details of the dictionary learning algorithm is described in [12].

Architecture Adaptation. $\mathcal{H}$ and $\mathcal{V}$ can be learned from $\mathcal{K}$ using an approximation algorithm as presented in [45]. The separation of overall convolutions allows us to convolve the input $\boldsymbol{x}$ with the vertical filter $\mathcal{V}$ to produce an intermediate feature map Z. Next, the desired output is generated by convolving $\boldsymbol{Z}$ with $\mathcal{H}$. This is implemented by replacing the original convolution layer with two successive convolution layers with filters $\mathcal{V}$ and $\mathcal{H}$ respectively.

4.4 Precision Scaling

Largely motivated by empirical observations, attention is increasing in how deep models can function surprisingly well even when represented by parameters with less numerical precision [22, 17]. Even with as few as 8-bits deep models for image or audio data have been observed to maintain relatively high accuracy levels. Experiments involving 16-bit and 8-bit are now becoming common, and binary types of network architectures are starting to evolve [18]. Promisingly hardware support to even further take advantage of these opportunities is increasing such as the Nvidia Tegra X1 [6] that has, for example, support for more efficient 16-bit operations within the GPU.

On similar lines, DXTK allows the developers to control the precision for model parameters, which directly impacts resources like the memory footprint of a deep model. In the current version of the toolkit, we offer the most basic forms of precision scaling – developers can change the model precision from double (64-bit) to float (32-bit), thereby reducing the model size by half. In the future, we will add more techniques to control the precision of the model parameters, including adjusting parameter precision to 16-bit and 8-bit. A number of other extensions of this Optimizer are possible. For example, quantization is a promising related direction that uses approaches like clustering or encoding schemes to more efficiently represent parameter values used within a model. Under one example technique of this type, K-means clustering is applied to assign a fixed-length code to each weight (requiring $\log_2 k$ bits) allow a weight matrix to be approximated by only storing k cluster means, in addition to each cluster index.

4.5 Parameter Cleaning

The deep models produced by frameworks like TensorFlow or Torch typically produce models that include parameters that are only at the training phase of the model, but never at inference phases. This is also true of models that are offered for download based on research. As a result, when using such frameworks or adopting a recent model published in the literature the number of parameters of the model is unnecessarily large for only running inference on and embedded or mobile device.

DXTK offers an Optimizer that performs this conceptually simple cleaning of unneeded parameters as a simple but useful way to reduce the memory footprint of deep models. This approach iterates through each layer of the model and removes parameters such as gradient weights and biases, which are computed during the backpropagation phase of model training and are no longer required at the time of inference. This approach can reduce the memory footprint of deep models by $\approx$ half, without accuracy loss.

5. EXPERIMENTS WITH DXTK

In this section we summarize the performance benefits, that DXTK brings, by thoroughly evaluating the techniques presented above. We consider two types of popular models, namely DNNs and CNNs, and measure compressed model performances for audio and image recognition tasks on four representative mobile and wearable platforms as shown in Figure 5: (i) Qualcomm Snapdragon 400, (ii) Nvidia Tegra K1, (iii) ARM Cortex M3 and (iv) ARM Cortex M0.

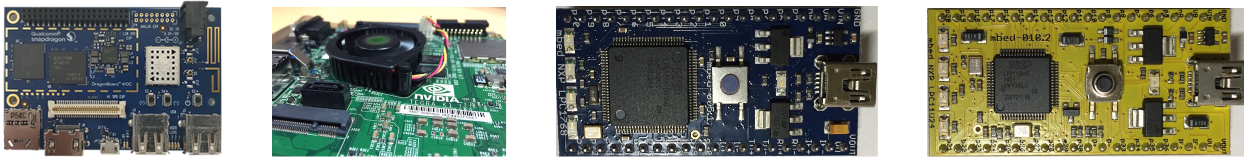

Figure 5: Hardware platforms used to evaluate DXTK. From left: Qualcomm Snapdragon 400, Nvidia Tegra K1, ARM Cortex M3 and ARM Cortex M0

5.1 Representative Deep Models

In this work we consider two DNN models, which are trained to identify the speaker from audio measurements and to understand the surrounding ambient environment. We next consider three CNN models, significantly rich in parameters than the DNNs, that identify object from a color image, predicts the gender and the age of a person from an input image. In the following we will briefly describe the models, and a summary of these models is presented in Table 1.

SpeakerID. The first DNN model is trained on an audio speaker verification dataset [48], which contains speech recordings from 106 individuals (45 male and 61 female). Audio measurements were recorded with 16 KHz sampling frequency. In order to maintain a near equal class distribution, we limit the maximum duration of audio recording to 15 minutes per user. SpeakerID is a DNN comprising of two hidden layers, each containing 1000 units and operates on aggregated mel-frequency cepstral coefficients (MFCC) extracted from the audio measurements. Input to the DNN is constructed by first extracting 13-dimensional MFCC features from 25 milli second measurement windows, followed by aggregating the features over a 5 second period. The aggregation generates an input feature dimension of 650. The trained DNN has over 1.7 million parameters.

Ambient. The second DNN model is trained on an audio scene dataset [1] that contains over 1500 minutes of audio recordings. All audio measurements were captured using Galaxy S3 smartphones and the dataset contains 19 different ambient scenes including, 'plane', 'busy street', 'bus', 'cafe', 'student hall' and 'restaurant'. During recording a sampling frequency of 22.05 KHz is used and several 30 seconds long audio files from each ambient environment are made available. In line with the SpeakerID model, we use the same MFCC-based feature extraction pipeline from the audio data and train a DNN with two hidden layers, each containing 1000 units.

AlexNet. We next consider a popular CNN model to show the benefits of using DXTK on a relatively recent computationally heavy image recognition task. AlexNet is an object recognition model [31], which supports more than 1000 object classes (e.g., dog, car). Compared to the DNN models described above, AlexNet has over 60.9 million tunable parameters. In 2012, it offered state-of-the-art levels of accuracy for well-known datasets like ImageNet.

AgeNet. Contrary to object detection task, AgeNet [37] is a CNN that tries to predict the age of a person from an input image. We use AgeNet in our experiments, which is composed of three convolution layers and two fully connected layers. The model has been trained using around $20K$ images, containing roughly equal proportion of male and female images.

GenderNet. Lastly, we consider another CNN model, GenderNet [37], that tries to recognize the sex of a person from an input image. GenderNet follows the same architecture as AgeNet (except the output layer) and uses the same dataset for training.

Name	Type	Parameters	Architecture
SpeakerID	DNN	1.8M	*fc*:3*
Ambient	DNN	1.7M	*fc*:3*
AlexNet	CNN	60.9M	*c*:5$^\imath$; *p*:3‡; *fc*:3*
AgeNet	CNN	11.4M	*c*:3$^\imath$; *p*:3‡; *fc*:3*
GenderNet	CNN	11.4M	*c*:3$^\imath$; *p*:3‡; *fc*:3*

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

Table 1: Representative Deep Models

5.2 Hardware Platforms

We report performances, viz. inference time and memory requirement of the deep models, while running inferences on four representative hardware platforms with various capabilities. Below we briefly describe the hardware platforms studied in this work. Note that, the toolkit support generic techniques, which bring inference time benefits (e.g., time and memory) to any underlying hardware platforms.

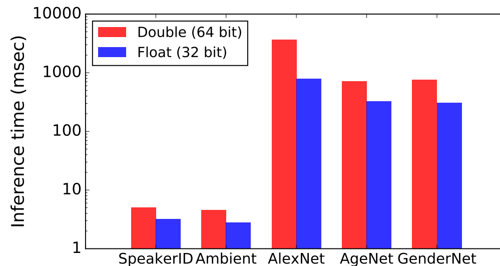

Figure 7: Inference times of various models under 64-bit (double) and 32-bit (float) precision on Qualcomm Snapdragon 400.

Qualcomm Snapdragon 400. Qualcomm Snapdragon 400 SoC [7] is widely found in many existing smartwatches, such as the LG G smartwatch R [4]. Internally, the Snapdragon has a quad-core ARM Cortex CPU and 1 GB of memory, though when shipped in smartwatches the RAM is often reduced to 512 MB. Figure 2 (left) shows the Snapdragon 400 development board used in our experiments.

Nvidia Tegra K1. Contrary to the other mobile and IoT hardware, Tegra K1 SoC [5] provides extreme GPU performance. It contains Kepler 192-core GPU, which is coupled with a 2.3 GHz 4-core Cortex CPU and an additional low-power 5th core (LPC) that is designed for energy efficiency. The K1 SoC is used in IoT devices such as June IoT Oven [3] and IoT-enabled cars. Mobile examples of the Tegra include the Nexus 9 along with the development smartphone in Google's Project Tango. The CPU can access over 1GB of RAM, largest of all processors profiled.

ARM Cortex M0 and M3. 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 $\mu W/MHz$ and support a memory size of 8 KB. The M3 variant of the cortex operates at 96 MHz and supports 32 KB of 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

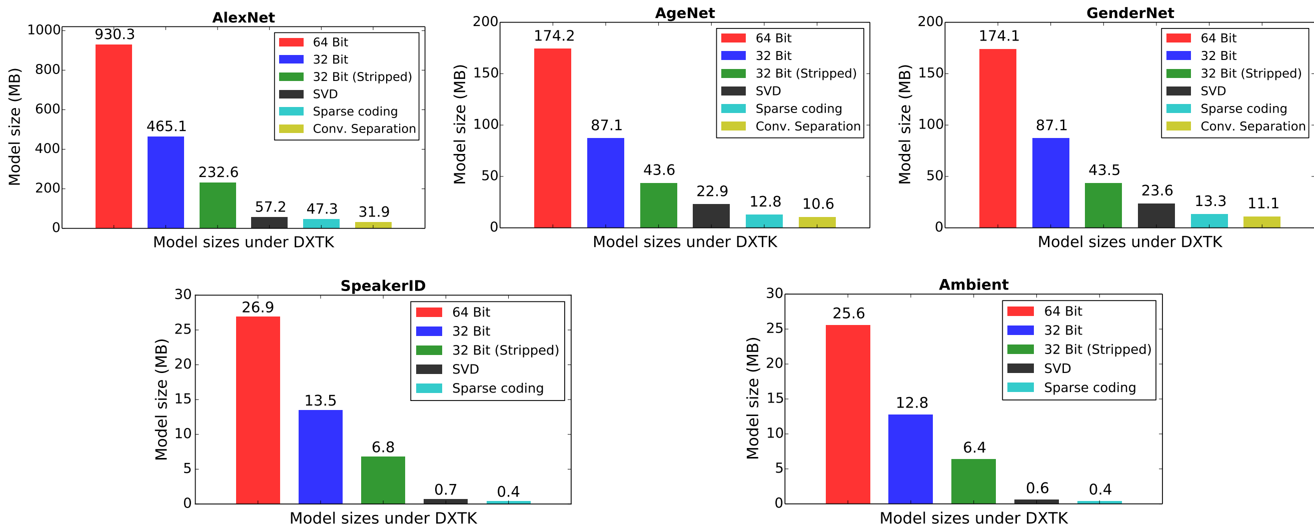

Figure 6: Memory footprint of all five models under various techniques supported by DXTK.

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 2: Summary of the Hardware Platforms

a small memory requires frequent paging, while executing a large model.

5.3 Performance Results

In this section we present performance benefits of DXTK, alleviating some of the bottlenecks in executing deep models on representative embedded platforms. We mainly focus on the memory requirement, model execution time and energy consumption.

Memory. We first demonstrate the effect of various optimization techniques on the runtime memory requirements of all five deep models. Figure 6 shows the memory footprint of the models in their original form (64 bit), after precision scaling (32 bit), in a stripped-down form (i.e., after parameter cleaning), then two forms of weight factorizations (WF and SF) when applied to the stripped-down model. Lastly, for CNNs (top row in Figure 6), we present the model sizes after performing convolution separation in addition to SF.

Clearly, we observe a drastic reduction in the memory footprint of the models when various optimization techniques supported by DXTK are applied to them - for example, the original AlexNet model size reduces from 930.3 MB to 31.9 MB (29x reduction) on applying the various optimisations, thereby making it feasible to run it on memory constrained hardware platforms. Similarly, the SpeakerID DNN model size reduces from 26.9MB to just 0.4MB (67x reduction) after optimization, which makes it possible to execute on processors as limited as Cortex M0.

Model Runtime and Energy Consumption. We now demonstrate the effect of the various optimization techniques on the runtimes of the models. Figure 7 shows the effect of precision scaling (64-bit to 32-bit) on all five models, when they are executed on the Snapdragon 400. We observe that changing the model precision alone can reduce the execution time significantly. For example, for SpeakerID a gain of $1.5\times$ is observed and the reduction was as high as $4.6\times$ for AlexNet (CNN).

Next, in Figure 8, we show the energy and latency of the 32-bit precision DNN models under various optimisation conditions on the four hardware platforms. We observe drastic reductions in runtimes of both models when SVD (WF) and sparse factorization (SF) are applied – for example, the fully optimized Ambient 32-bit DNN model runs $9.7\times$ faster (on Cortex M0) and $4.9\times$ faster (on SnapDragon 400) than its unmodified version. Similar reductions in the energy consumed by these models are observed across all hardware platforms – for example, the fully optimized SpeakerID model consumes $9\times$ less energy on Cortex M0 and $2\times$ less energy on SnapDragon 400.

Similarly, Figure 9 shows the energy and latency profile for the three CNN models only for Snapdragon 400 and Tegra K1, as the Cortex M0 and M3 processors are not capable of running the CNNs in reasonable time. In the case of CNN models, we also apply the convolution separation technique, which further reduces the overall inference time. For example, we observe that the fully optimized AlexNet model consumes $2.3\times$ less energy and runs $2.3\times$ faster than unmodified AlexNet on Nvidia Tegra K1.

Inference Accuracy. DXTK enables developers to reduce the memory footprint of a deep model significantly on embedded devices, without compromizing significantly on the inference accuracy. More specifically, DXTK seeks to constrain accuracy loss to 5% or less (compared to the original model before any changes are made). Table 3 illustrates the relative accuracy loss for SpeakerID (DNN) and AlexNet (CNN) under different memory reduction conditions. For example, in the case of AlexNet, the model size can be reduced by more than 75% with just a 4.9% accuracy loss. DXTK also has the ability to adjust the model size dynamically based on the memory availability and as such, can be deployed to new hardware platforms with different memory sizes without requiring system changes.

	Relative Accuracy Loss (%)	Memory Reduction (%)
SpeakerID	3.2 (93.7 to 90.5)	92.8 (28 MB to 2 MB)
AlexNet	4.9 (77.5 to 72.6)	75.5 (233 MB to 57 MB)

Table 3: Model size reduction relative to accuracy loss, when applying the estimator threshold.

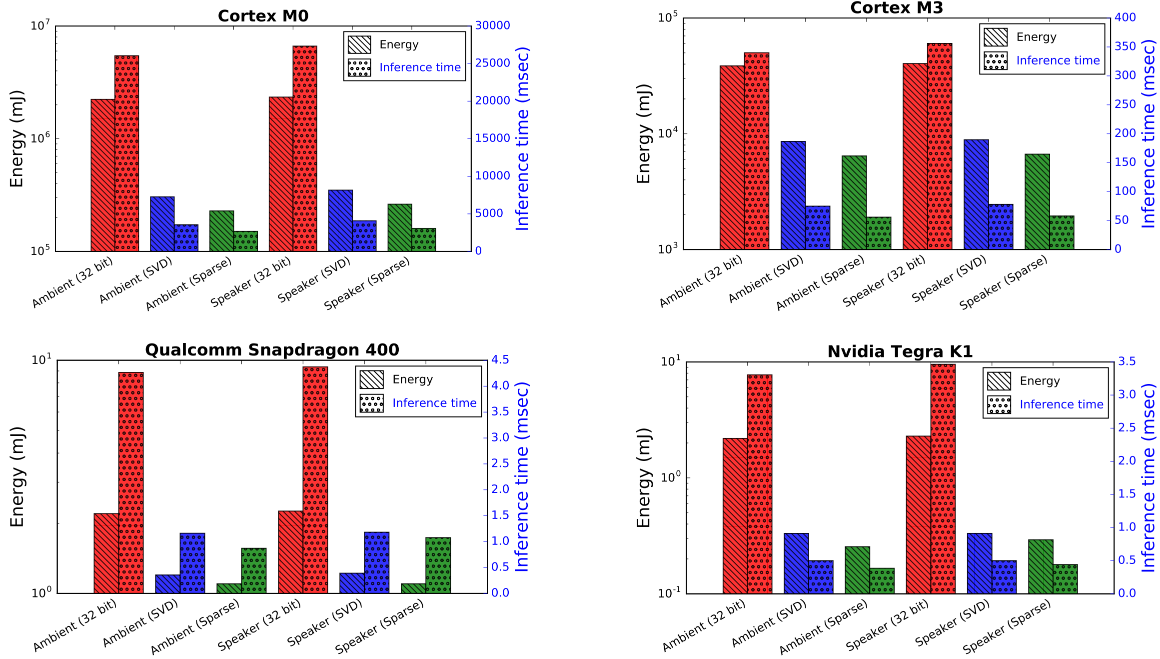

Figure 8: Model runtime and energy consumption when running two DNNs on ARM Cortex M0, Cortex M3, SnapDragon 400 and Tegra K1

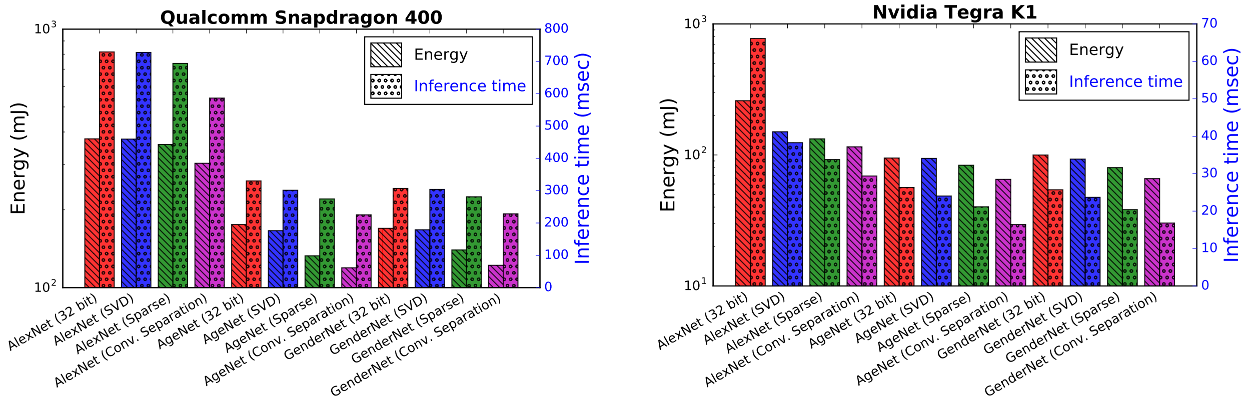

Figure 9: Model runtime and energy consumption when running three CNNs on SnapDragon 400 and Tegra K1

6. DISCUSSION

We now provide a discussion on the current state and future extendability of the DeepX toolkit. We also highlight the alternate viewpoint in the industry on bringing deep learning onto the embedded devices.

First Release of DXTK. It is important to stress that the toolkit version presented in this paper is a first version wherein it supports deep learning models developed in Torch and a limited number of optimizations. We believe that these initial techniques provide a good starting point for users to prototype and deploy deep inference applications on embedded devices, or to test the performance of their inference models directly on the device.

We are actively working on adding several extensions to the toolkit. We have already developed better optimization techniques for deep models that we will be releasing in the future versions. For instance, one key optimization to be added in the future versions is the ability to decompose the monolithic deep model network architecture into unit-blocks of various types, that are then more efficiently executed by heterogeneous local device processors (e.g., GPUs, CPUs). Further, as more open source deep learning frameworks (e.g. TensorFlow) become popular, DXTK can be extended to support the models developed in them. The modular nature of DXTK ensures that it can support a variety of deep learning frameworks and optimisation techniques in the future – and its users can choose their preferred framework and techniques.

Dedicated Hardware for Deep Learning. Recently, there has been a push to develop dedicated hardware to run deep learning algorithms [16, 51]. However often these prototypes perform a specific type of deep learning (such as a CNN) or only certain types of deep model layer types (e.g., a convolution), with remaining layers executed as normal. Moreover dedicated hardware are expensive and not as widely deployed in modern smartphones and embedded devices. As such, we argue that for devices that do not run on dedicated deep learning hardware, it become essential to apply software optimisation techniques, some of which are currently incorporated in DXTK and others that will be added in the future. Even on the dedicated hardware, DXTK provide ways for effective model

compression and in future will support intelligent model decomposition and parallel computation across processors – thus adding to the acceleration and performance improvements provided by the dedicated hardware. In summary, we believe that a combination of better hardware and extendable software toolkits like DXTK could be the way forward for mobile deep learning.

7. RELATED WORK

We now overview work closely related to DXTK, this includes recent advancements in embedded deep learning and efforts to optimize sensing algorithms for constrained devices.

Mobile and Embedded Applications of Deep Learning. Only recently has the exploration into deep learning methods for mobile and embedded scenarios begun (e.g., [34, 23]). Deep learning models have been trained for a range of inference tasks on mobile devices, such as speaker identification [35], ambient scene analysis [35], emotion classification [24], garbage detection in images [38], and activity recognition [50]. Google has enabled forms of its deep learning translation models to run directly on a phone [9], and deep learning has already revolutionised mobile speech recognition in commercial systems [27]. But these few examples rely on manual per-model optimizations, provided by teams of people with high-levels of expertise in deep learning and mobile devices. In contrast, DXTK aims to allow any developer to use deep learning methods and automatically lowers resource usage to levels that are feasible for mobile and embedded devices.

Further, deep learning frameworks such as Torch and TensorFlow are being ported for mobile operating systems such as Android, iOS and Linux. This has allowed developers to use off-the-shelf models trained in these frameworks and execute them on mobile and embedded platforms. The various optimizations in DXTK are currently supported for the deep models trained using Torch, and in future we plan to add support for more deep learning frameworks including TensorFlow.

Optimization Techniques for Embedded Devices. The resource constrained nature of embedded devices has motivated extensive research on developing techniques to run context-sensing and behavior modeling algorithms under constraints of memory and processing power. 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 [29, 40, 41]. Other works such as [39] have looked at combining such optimizations with careful usage of energy efficient hardware.

Similar research is also increasing for lowering resource consumption in deep models – for example, the *model compression* research in the machine learning community focuses on altering the underlying model at training time in order to scale it to the target hardware. For example, [26] actually removes nodes and reshapes layers within the model while [21, 42] performs types of quantization of parameters within layers. On the contrary, we designed DXTK towards minimizing the modifications made to the model and so adopt approaches that insert new layers designed to optimize performance.

Researchers have also demonstrated one-off optimizations such as [35] that scaled down DNNs to run directly on a DSP only, offering energy efficiency. Others have proposed smaller footprint deep models for tasks such as keyword spotting or speaker verification – these models are much smaller than normal and so can run on phones [15, 47]. DXTK instead targets full-scale deep models that otherwise only appear in cloud systems, and enable them to be executed on embedded devices.

Hardware specialization is another promising direction for deep learning optimization with many studies already underway [16, 51]. However often these prototypes perform a specific type of deep learning (such as a CNN) or only certain types of deep model layer types (e.g., a convolution), with remaining layers executed as normal. Furthermore, we also expect DXTK will leverage specialist hardware as they become more available.

8. CONCLUSION

In this work, we have presented the design and evaluation of the DeepX toolkit (DXTK) – a collection of tools designed to assist the exploration and adoption of deep learning methods on embedded and mobile devices. Although DXTK includes a number of auxiliary components such as ready-to-use deep models prepared for low-resource environments, the technical focus of the toolkit is a series of Model Optimizers that modify deep models to reduce their usage of resources such as memory or computation. In our initial release described here, these techniques include: matrix and sparse factorization, convolution separation, precision scaling and parameter cleaning. To demonstrate the value this toolkit, we report example performance experiments of DXTK across a variety of deep learning models including DNNs and CNNs. Results show that DXTK can significantly reduce the memory footprint of a deep model with little loss in inference accuracy. This is significant for enabling their execution on resource-constrained embedded devices, and providing a much higher inference accuracy than the conventional shallow models currently in use. We hope that the techniques and supporting tools embodied by DXTK will help accelerate the adoption of deep learning within constrained devices, and then enable further research in the area.

9. REFERENCES

[1] Ambient Audio Dataset. https://sites.google.com/site/alainrakotomamonjy/home/audio-scene.

[2] DeepX Toolkit. http://deepx.tech/.

[3] June Oven. http://juneoven.com/.

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

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

[6] Nvidia Tegra X1. http://www.nvidia.com/object/tegra-x1-processor.html.

[7] Qualcomm Snapdragon 400. https://www.qualcomm.com/products/snapdragon/processors/400.

[8] Torch. http://torch.ch/.

[9] How google translate squeezes deep learning onto a phone. http://googleresearch.blogspot.co.uk/2015/07/how-google-translate-squeezes-deep.html, 2015. Accessed: 2016-04-10.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[11] Y. Bengio, I. J. Goodfellow, and A. Courville. Deep learning. Book in preparation for MIT Press, 2015.

[12] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, SenSys '16.

[13] 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*, 15:242–262, 2014.

[14] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[15] 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*, ICASSP'14, 2014.

[16] 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*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[17] M. Courbariaux, Y. Bengio, and J. David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[18] M. Courbariaux, Y. Bengio, and J. David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015.

[19] L. Deng and D. Yu. Deep learning: Methods and applications. Technical Report MSR-TR-2014-21, January 2014.

[20] T. Deselaers, S. Hasan, O. Bender, and H. Ney. A deep learning approach to machine transliteration. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 233–241. Association for Computational Linguistics, 2009.

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

[22] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[23] 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.

[24] K. Han, D. Yu, and I. Tashev. Speech emotion recognition using deep neural network and extreme learning machine. 2014.

[25] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 123–136, New York, NY, USA, 2016. ACM.

[26] 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*, pages 245–249. IEEE, 2014.

[27] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[28] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[29] 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*, pages 267–280. ACM, 2008.

[30] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[32] 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*.

[33] 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*, pages 7–12. ACM, 2015.

[34] 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*, pages 117–122. ACM, 2015.

[35] 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*, UbiComp '15, pages 283–294, New York, NY, USA, 2015. ACM.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[37] G. Levi and T. Hassner. Age and gender classification using convolutional neural networks. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) workshops*, 2015.

[38] G. Mittal, K. B. Yagnik, M. Garg, and N. C. Krishnan. Spotgarbage: Smartphone app to detect garbage using deep learning. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, pages 940–945, New York, NY, USA, 2016. ACM.

[39] 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*, pages 83–94. ACM, 2015.

[40] 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*, pages 29–42. ACM, 2012.

[41] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[42] J. S. Ren and L. Xu. On vectorization of deep convolutional neural networks for vision tasks. *arXiv preprint arXiv:1501.07338*, 2015.

[43] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[44] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

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

[46] 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.

[47] 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*, pages 4052–4056. IEEE, 2014.

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

[49] J. Xue, J. Li, and Y. Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, 2013.

[50] M. Zeng, L. T. Nguyen, B. Yu, O. J. Mengshoel, J. Zhu, P. Wu, and J. Zhang. Convolutional neural networks for human activity recognition using mobile sensors. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 197–205. IEEE, 2014.

[51] 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*, pages 161–170. ACM, 2015.