Exploring the Capabilities of Mobile Devices in Supporting Deep Learning

Yitao Chen
ychen404@asu.edu
Arizona State University

Saman Biookaghazadeh
sbiookag@asu.edu
Arizona State University

Ming Zhao
mingzhao@asu.edu
Arizona State University

ABSTRACT

Deep neural networks (DNNs) have unleashed a new wave of applications on mobile devices, such as various intelligent personal assistants. Most of these applications rely on the use of cloud resources to perform deep learning. With increasingly more powerful mobile devices, users can perform more deep learning tasks on the devices. In addition, learning on the devices has important advantages, such as personalization, privacy, and responsiveness; however, a good understanding of the capabilities of modern mobile devices in supporting deep learning is generally lacking. To address this gap in knowledge, this paper presents a comprehensive study on performing training and inference on mobile devices. It develops TensorFlow+, an extension of the widely used TensorFlow framework, to enable training DNNs on devices and use the available GPUs to accelerate the learning tasks. The study focuses on four aspects: 1) the performance impact of the network architecture; 2) the effectiveness of using accelerators for learning on mobile devices; 3) the resource and battery usages of training and inference; and 4) the performance impact on other applications running on the devices. The results show that the size (width and depth) of a network as well as the types of layers that it uses are important to not only meeting the device's capability but also to the performance of learning. The study also shows that hardware acceleration is important to both improving the speed of learning and reducing the impact on other applications on the device.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Heterogeneous (hybrid) systems**; **Embedded software**.

KEYWORDS

Deep learning, neural networks, edge computing, mobile computing

ACM Reference Format:
Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. 2019. Exploring the Capabilities of Mobile Devices in Supporting Deep Learning. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3318216.3363316

SEC '19, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363316

1 INTRODUCTION

With the rapid advancement of mobile processors, users can accomplish a significant amount of daily tasks on mobile devices. In particular, deep learning has unleashed a new wave of applications, such as augmented reality, image classification, and face recognition on mobile devices, using computationally expensive models such as deep neural networks (DNNs). Such models are usually trained using scalable cloud resources for hundreds to thousands of hours.

There are several serious drawbacks with such a cloud-only deep learning approach: 1) the mobile devices often have to rely on off-the-shelf pre-trained models, limiting the models' adaptation to the users' local inputs; 2) users need a reliable network connection to provide inputs to and obtain results from the models trained in the cloud; and 3) the rapid growing number of devices and data that they collect will soon outgrow even the capacity of the cloud systems.

In comparison, there are important benefits of using mobile devices for deep learning: 1) *Personalization*—users can tailor models with locally available data to better satisfy users' current needs; 2) *Privacy*—the data used for training a tailored model can be better protected if it is in the owner's device instead of on the shared resources in a cloud; 3) *Responsiveness*—learning on devices can provide more prompt results, without being affected by the possible outages of the cloud or unreliable connections to the cloud.

Researchers and developers are actively studying new techniques to enable learning on mobile devices, but a good understanding of the capabilities of such devices to support learning is generally lacking. For example, *model compression* techniques [6, 18, 18, 23, 29] can reduce the size of a model to fit on a mobile device. Another approach is *knowledge transfer*, which trains smaller, on-device networks under the supervision of larger, in-cloud networks [3, 38, 40, 44]. However, these works focus on the algorithm aspect of deep learning and do not fully explore how well such algorithms work on mobile devices. Prior works [24, 36] studied performing inference of neural networks on mobile platforms, but they did not consider training and did not fully explore the capabilities of modern mobile devices.

In this paper, we investigate the software and hardware capabilities of mobile devices to support deep learning algorithms. Our study focuses on the following four main aspects: 1) the impact of the network architecture (width, depth, and different types of layers) on learning on devices; 2) the effectiveness of using accelerators available on devices to help the learning; 3) the impact of learning on the resource and battery usages of a device; and (4) the impact on the performance of other applications running on the same device.

Since none of the popular deep learning frameworks support training on mobile devices, we extended TensorFlow (version r1.3) [1], a widely used framework, to enable training. The vanilla TensorFlow for Android allows only inference; our extension enables it to train DNNs on Android platform, as well. We also extended TensorFlow to accelerate learning using the available GPU on a device, which allows TensorFlow to use RenderScript [17] to take advantage of heterogeneous hardware and accelerate both inference and training. In comparison to previous work [2], we significantly improved TensorFlow to enable the acceleration of both training and inference, and performed a thorough study on the effectiveness of acceleration.

We focus on DNNs designed for image classification tasks which have shown impressive capability ever since AlexNet [26] won the ILSVRC 2012 [39]. We studied models that represent the aforementioned two possible approaches to utilizing mobile devices in deep learning: 1) MobileNet [20], a compressed model designed to fit to a mobile device's resource constraints, and 2) Mentee network [19], a small network that can run on a device and receive supervision from a mentor network in the cloud. We also studied ResNet [45], a popular model for learning on server platforms. Our test platforms include several generations of mobile devices, Pixel 2, Nexus 7, Nexus 5, and an Internet of Things (IoT) platform, Raspberry Pi 3B+.

The most significant results of our study are as follows: 1) The size (width and depth) of a network and the types of layers that it uses are important to not only meeting the device's capability but also to the performance of learning. Models based on convolutional layers are more sensitive to the depth whereas models based on fully-connected layers are more sensitive to the width. Moreover, convolutional layers are substantially more expensive (two orders of magnitude slower) than fully-connected layers on the device; 2) Hardware acceleration is important to improving learning speed and reducing the impact on other applications on the device. By using the device's GPU to accelerate both the forward and backward paths of deep learning, our extended TensorFlow can cut down the training time by 44.8%. Overall, we conclude that using mobile devices to support deep learning is feasible, but we need to pay attention to network architecture and make good use of the available accelerators to speed up the training.

To the best of our knowledge, our work is the first to provide a comprehensive study of deep learning on mobile devices. The contributions in this study are as follows: (1) extend commonly used deep learning frameworks such as TensorFlow to support training on mobile devices; 2) enable the use of GPUs to accelerate both training and inference on mobile devices; (3) provide an in-depth examination of the capabilities of mobile devices for supporting training and inference using DNNs; and (4) quantitatively analyze the impact of learning on mobile devices to resource usages and user experience.

The rest of the paper is organized as follows: Section 2 introduces the background and motivations; Section 3 describes the methodology for our study; Section 4 presents the analysis of the experimental results; and Section 5 concludes the paper.

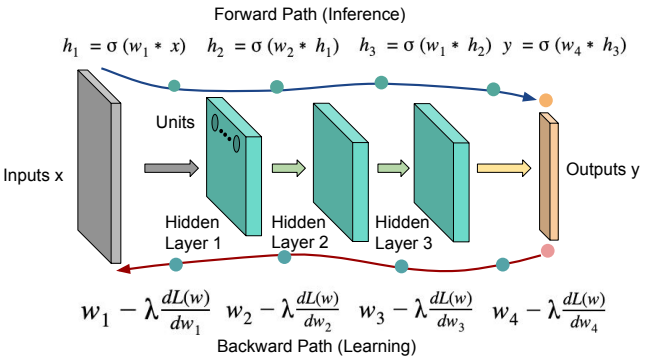


Figure 1: The forward path and the backward path of a four-layer CNN [46]. $w(\cdot)$ denotes the weights of each layer. $w * x$ denotes a convolution operation between w and x. λ is the learning rate. $\sigma(\cdot)$ is the activation function. L is the loss to be optimized, which is the difference between the prediction and ground truth label.

2 BACKGROUND AND MOTIVATIONS

2.1 Deep Learning and Mobile Computing

Deep learning is the key to various mobile computing tasks, such as voice recognition, natural language processing, image classification, and object detection. DNNs have achieved remarkable accuracy in the above applications compared to other machine learning algorithms. Image classification is one of the major applications of deep learning algorithms, which is computationally intensive and involves a large volume of data. Such tasks obtain images as input and use neural networks to label each image with a class that the image belongs to.

A neural network consists of multiple connected layers. A layer is an abstraction to help with the modular design of a network, where each layer performs a specific mathematical operation. Examples of common layers include *fully-connected layers*, which connect to all neurons from the previous layer and output a weighted sum, and *convolutional layers*, which use kernels with various sizes to extract the features from the overlapped area of the input.

Deep learning tasks include *training* a model and using the model to perform *inference*. The training follows the backward path of a network and updates the weights of each layer iteratively towards a target. The inference follows the forward path and uses the trained weights of each layer to make a prediction based on the input. In the context of image classification, training updates the weights of a model to reduce the difference between the prediction and the ground truth label, whereas inference uses images as input to predict the image class. Figure 1 illustrates the forward path and the backward path of a four-layer convolutional neural network (CNN). The backward path calculates the partial derivatives of the previous layers' input, which involves more calculation compared to the forward path. Besides, each layer needs to store the intermediate results to complete the calculation of partial derivatives. Hence, the backward path calculation is more demanding on both computation and storage compared to the forward path.

The rapid advancement in mobile computing technologies is enabling more learning-based applications. These applications are usually computational- and data-intensive but mobile devices have

only limited resources. The processors on a device are designed for high power efficiency, rather than high performance. The available memory is quite limited due to the device's small form factor. The devices are powered by batteries with limited capacity. Therefore, a good understanding of the capabilities of mobile devices for deep learning is vital for many ongoing and future research/development efforts in edge and mobile computing.

2.2 Learning on Mobile Devices

The results from this paper can benefit a number of related efforts on enabling deep learning on mobile and embedded devices. We can broadly classify these works into the following three categories:

Model compression. There are several compression techniques for reducing the complexity of DNNs and allowing them to be used for inference on devices, including (1) *weight sharing*, which reduces the memory footprint of network parameters by using a single value to represent a group of weights [6, 18]; (2) *pruning*, which reduces the complexity of a model by eliminating the weights under a pre-determined threshold [29]; (3) *quantization*, which reduces the size of a model by shrinking the number of bits used to represent the weights [23]; and (4) *encoding*, which encodes the intermediate layer output to reduce memory footprint [22].

Knowledge transfer. A large network (*Mentor*) trained in the cloud can transfer its learned representative features to a small network (*Mentee*) trained on the device, thereby improving the latter's accuracy and convergence speed. Knowledge transfer techniques are inspired by the Dark Knowledge (DK) work [19], which distills information from the softmax layer output. Romero et al. [38] extended the DK approach to include the intermediate layer from the Mentee network for knowledge transfer. Sharma et al. compared the effectiveness of different knowledge transfer techniques [40]. DeCAF [7] is a related solution that retrains only the last layer of a pre-trained model to adapt to new learning tasks.

Federated learning [25, 42] is a solution for training a large, global model on a centralized server by federating a number of small, distributed models trained on mobile devices. Each mobile device acts as a node and calculates an update from its small model using its local data. This node then sends this update to the global model. Federated learning can reduce the risk of exposing users' data, since updating the global model is considered less privacy-sensitive compared to sharing the input data globally.

The above related works focused on the algorithm aspects of involving mobile devices in deep learning but did not fully explore how well such algorithms work on real devices. There are also several related works that explored only the use of inference on mobile devices. Lane et al. proposed an inference engine for mobile sensing [28]. M. Alzantot et al. and S. Rallapalli et al. [2, 36] studied the use of on-device GPUs to accelerate inference. Neurosurgeon [24] considered partitioning the layers of a network between device and cloud and then optimizing it for the inference speed and energy usage. Chen et al. [5] presented a compiler for deploying deep learning on different hardware backends, including mobile phones. These studies did not consider training, which is a lot more challenging than inference and is becoming increasingly important to many mobile applications [16]. Moreover, these studies did not fully consider the various capabilities of modern mobile devices for supporting deep learning. Therefore, a comprehensive study is much needed to fully explore all these important aspects of deep learning on mobile devices. The rest of this paper presents our findings.

Table 1: Comparison of deep learning frameworks on mobile devices

	Inference	Training	Acceleration
TensorFlow Mobile	√	×	×
TensorFlow Lite	√	×	×
PyTorch	√	×	×
Deeplearning4j	√	√	×
MXNET	√	×	×
Chainer	×	×	×
MLib	×	×	×
CNTK	×	×	×

3 METHODOLOGY

3.1 Porting TensorFlow to Mobile Devices

To perform this study, we need a framework that supports both training and inference on mobile devices. Several widely used deep learning frameworks have versions designed for mobile devices, such as TensorFlow Mobile [15], TensorFlow Lite [13], and Caffe 2 [10], but they support only inference using a pre-trained network. Table 1 compares the popular frameworks on the capability of performing deep learning tasks on mobile devices. Most of the frameworks support only inference on mobile devices. We decided to extend TensorFlow Mobile to also support training.

TensorFlow uses a data flow graph to represent the computation. In a data flow graph, a node represents an operation and an edge represents the data used or generated by the next or previous computation. Multiple operations form a layer, a core abstraction in deep learning frameworks, for modularity in the design. Figure 2 represents the general architecture of TensorFlow (left) and the data flow graph for training a model (right). TensorFlow consists of multiple layers. At the bottom, it provides different implementations for all operations using various device programming models, which enables acceleration on different types of processors. Further, it provides several types of operations required for training a DNN. On the top, TensorFlow provides various interfaces to leverage the core functionalities. In our case, we utilized the Java interface and extended the Android library to support training. The right part of the diagram demonstrates the data flow of the training process, which consists of forward and backward computations. In the forward path, layers call normal computational kernels (Ⓐ, Ⓑ), and in the backward path, they call the gradient calculation functions (Ⓒ, Ⓓ, Ⓔ) of those kernels.

TensorFlow Mobile only provides an interface to support inference-related operations. In order to train deep learning models on the device, we extended the interface to support training as well. We need to first, modify the Java interface to support training-related operations and second, add all the missing libraries that are needed for training to the Android library. The training process involves more operations than that of the inference process. Different

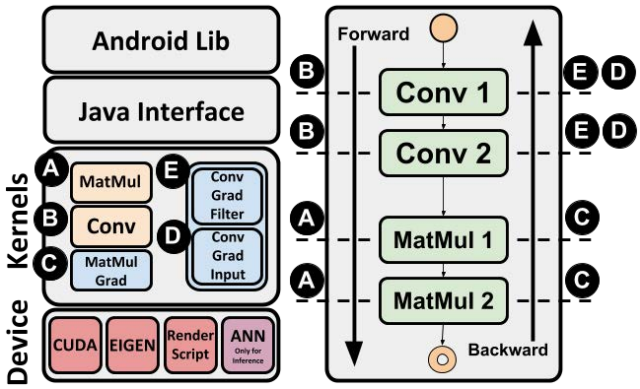


Figure 2: The general architecture of TensorFlow and the flows of the forward and the backward paths.

from inference, the training process needs to perform operations without generating any prediction, such as variable initialization. To support all the training related operations, we modified the Run() function in the Java interface to perform operations without producing a prediction. We used the modified Run() function to initialize all the variables and perform the training related operations. We then added the missing kernels to TensorFlow Mobile, by porting them from the TensorFlow C++ core to Android.

We call our extended framework described above ***TensorFlow+***. It utilizes model graphs defined in Python and provides the same interface for training on devices as that of TensorFlow on the server. Model graphs use Protobuf format without freezing any variables. TensorFlow+ loads the model graph with the TensorFlow Java interface. It then loads the dataset images batch by batch from the device. Using our training-enabled Java interface, TensorFlow+ can train a network model on devices following the same steps as how TensorFlow works on a server.

3.2 Accelerating Inference and Training

We further improved our solution to leverage the available accelerators, such as GPUs, on mobile devices, for accelerating deep learning operations. The existing mobile version of TensorFlow uses only CPUs on Android platforms. It relies on the Eigen library [9], which is a high-level C++ library optimized for linear algebra operations and related algorithms. But as the network models become increasingly more complex (deeper and wider), such a CPU-based approach becomes inefficient. At the same time, the increasingly available accelerators on mobile devices make it possible to use them for accelerating deep learning on devices.

Our general approach is to use the RenderScript [12] framework to implement the computationally expensive operations in deep learning. RenderScript parallelizes the workload at runtime on different processors using a language derived from the C99 [37] standard. There are other options to leverage GPUs in mobile devices, such as OpenGL ES and OpenCL. But OpenGL ES is for graphical tasks; in order to use it to accelerate computational tasks, we need to implement all the functionalities, including matrix multiplication. OpenCL, however, is not officially supported by Android. Previous work RSTensorFlow [2] took the same approach as ours to accelerate matrix multiplication and convolution operation in inference which involves only the forward path of DNNs. But it did not produce good speedup for convolutions (which accounts for around 75% of the forward path time). It also does not support the acceleration of the backward path of DNNs, which is the most intensive component in training and, according to our study, accounts for 70% of the total training time.

To accelerate the backward path of DNNs, we need to accelerate the gradient calculations. In TensorFlow, gradients are calculated in two separate kernels, *conv_grad_input* and *conv_grad_filter*, which calculate the gradients with respect to the input and the filter. The major calculations involved in these two kernels are matrix-matrix operations. In TensorFlow+, we replaced their Eigen-based implementation with RenderScript's single-precision matrix-multiplication (SGEMM) in order to accelerate the gradient calculations. The SGEMM API performs operation $C = \alpha \times op(A) \times op(B) + \beta \times C$, where A, B, and C are matrices, α and β are parameters. In the *conv_grad_input* kernel, computation involves three matrices: the input, the filter, and the output. TensorFlow+ redirects the matrices in the *conv_grad_input* kernel to the SGEMM implementation to speedup the calculation. Similarly, TensorFlow+ also uses the SGEMM API to speed up the *conv_grad_filter* kernel. In addition to GPUs, we also considered to use the recently released Pixel Visual Core (PVC), a specialized image processor, to accelerate learning on mobile devices. However, the only API provided by Android to access this accelerator—the Android Neural Network (ANN) API, supports only inference, but not training.

In summary, we ported the widely used deep learning framework, TensorFlow, to Android, with the support for both training and inference, and improved their performance by using available accelerators (GPU) on mobile device. In the next section, we present a comprehensive evaluation of our solution TensorFlow+ and analysis of the capabilities of mobile devices in supporting deep learning.

4 EVALUATION

4.1 Setup

Models. We considered network architectures that are suited for mobile devices, as commonly used DNN models such as VGG16 are too large for the memory size of the devices. So we first studied an architecture based on the *Mentee network*, proposed for knowledge transfer [19], which has five convolutional layers and three fully-connected layers with around 26.89 million parameters in total. We also considered *MobileNet* [20], a widely used network model designed for vision applications on mobile and embedded devices [41]. However, this network still cannot fit in our test devices for training, so we shrank the base MobileNet model to a six-layer model (with around 5.48 million parameters in total) at the cost of losing some accuracy (around 1% in the Top-5 accuracy and 9% in the Top-1 accuracy compared to the original model). Similarly, we also shrank the ResNet [45], a popular model for server platforms, to an eight-layer model so that it can fit in our test devices. Table 2 lists the detailed architecture of the models.

Datasets. We investigated different commonly used datasets and realized that large inputs tend to force the batch size to small values, due to the limited memory. So we chose *CIFAR-10*, which consists of

Table 2: Architecture of the Mentee, reduced MobileNet, and reduced ResNet models. the *Configuration* column shows the detailed parameters of each layer. Convolutional layer parameters are denoted as $D_H \times D_W \times M \times N$, where D_H and D_W denote the height and width of a kernel, M denotes the number of input channels, and N denotes the number of output channels. Average pooling layer parameters are kernel dimensions. Fully-connected layer parameters are the output dimensions.

Mentee Network		Reduced MobileNet	
Type	**Configuration**	**Type**	**Configuration**
Conv	3×3×3×64	Conv	3×3×3×32
Conv	3×3×64×128	Conv/dw	3×3×32
Conv	3×3×128×256	Conv	1×1×32×64
Conv	3×3×256×512	Conv/dw	3×3×64
Conv	3×3×512×512	Conv	1×1×64×128
Avg Pool	2×2	Conv/dw	3×3×128
FC	4096	Conv	1×1×128×128
FC	4096	Conv/dw	3×3×128
FC	10	Conv	1×1×128×256
Softmax	Classifier	Conv/dw	3×3×256×256
-	-	Conv	1×1×256×256
-	-	Avg Pool	7×7
-	-	FC	10
-	-	Softmax	Classifier

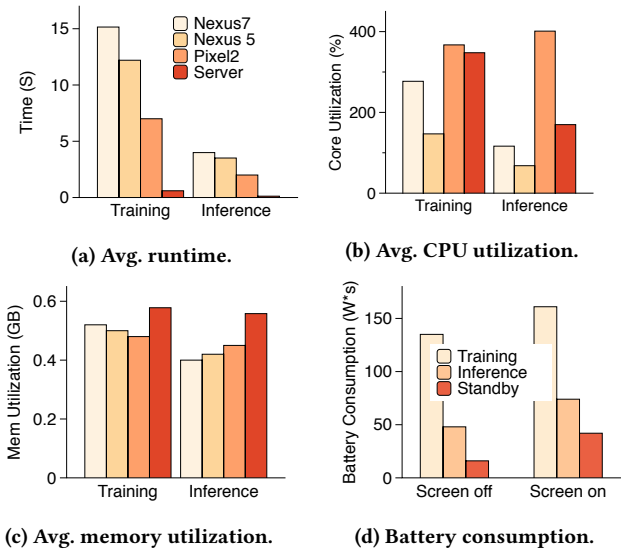
Reduced ResNet	
Type	**Configuration**
Conv	3×3×3×16
Conv	3×3×16×160
Conv	3×3×160×320
Conv	3×3×320×320
Conv	3×3×320×640
Conv	3×3×640×640
Avg Pool	8×8
FC	10×10

60,000 32×32 images and is widely used in deep learning works [11, 21, 30, 43], as the benchmark dataset. With CIFAR-10, we are able to conduct our experiments with a reasonable batch size of 128.

Devices. We obtained our results on several different generations of mobile devices, Nexus 7 (released in 2012), Nexus 5 (released in 2013), and Pixel 2 (released in 2017). We chose these devices because they represent different generations of devices. For example, Pixel 2 has similar hardware specifications as iPhone 8 and Samsung Galaxy S8. Pixel 2 was released in October 2017 based on the Qualcomm Snapdragon 835 SoC. It has an eight-core, Qualcomm Kryo 280 CPU running at 2.45 GHz clock speed and 4 GB LPDDR4 RAM at 1866 MHz. Apple iPhone 8 was released at a similar time, equipped with an A11 Bionic chipset with 2.39 GHz hexa-core (2 × Monsoon + 4 × Mistral) and 2133 MHz LPDDR4 RAM. Both Pixel 2 and iPhone 8 have similar specifications in terms of CPU speed and memory speed. We also performed experiments on a Raspberry Pi 3B+, a commonly used Internet of Things (IoT) platform, to explore the capability of the emerging IoT devices in supporting deep learning. As a baseline, we also obtained results from a typical datacenter server. Table 3 lists the hardware specifications of our test platforms.

Table 3: Test Platform Specifications

Specifications		
Pixel 2	OS	Oreo 8.1.0
	CPU	2.45 GHz Octa-core Kryo
	GPU	710 MHz Adreno 540
	Visual Core	800 MHz Cortex-A53
	Memory	4 GB
Nexus 7	OS	Marshmallow 6.0.1
	CPU	1.5 GHz quad-core Krait 300
	GPU	400 MHz Adreno 320
	Memory	2 GB
Nexus 5	OS	Marshmallow 6.0.1
	CPU	2.26 GHz quad-core Krait 400
	GPU	450 MHz Adreno 330
	Memory	2 GB
Resberry Pi 3B+	OS	Rasbian
	CPU	1.4 GHz quad-core Cortex A53
	Memory	1 GB
Server	OS	Ubuntu 16.04 LTS
	CPU	Dual Intel Xeon E5-2630
	GPU	Nvidia Tesla K40
	Memory	64 GB



(a) Avg. runtime. **(b) Avg. CPU utilization.**

(c) Avg. memory utilization. **(d) Battery consumption.**

Figure 3: Mentee network's performance and resource usages.

4.2 Overall Performance and Resource Usages

We first study the performance and resource usages of using TensorFlow+ to train the Mentee, MobileNet, and ResNet networks on mobile devices (*without* using accelerators).

Figures 3a shows the runtime of using the Mentee network to perform training and inference on different platforms. The batch size is 128 for both training and inference. The results from training are for only a *single* iteration of one batch of images. We can observe substantial performance improvement with the continuous advancement of mobile platforms. Interestingly, the performance gap between the devices and server is more significant for inference than for training. For example, a high-end device like Pixel 2 takes 10× longer time for training than the server and 15× longer for inference.

Figures 3b and 3c show the average CPU and memory utilization of the above training and inference experiments. The results confirm that both training and inference are CPU intensive. Their average memory utilization during training and inference is not high. Both training and inference do not need to perform frequent memory allocations and the major part is in the initialization where the system needs to allocate memory for storing the parameters of a model and the batch of images. Then, the CPU performs calculations until the batch of images is fully consumed.

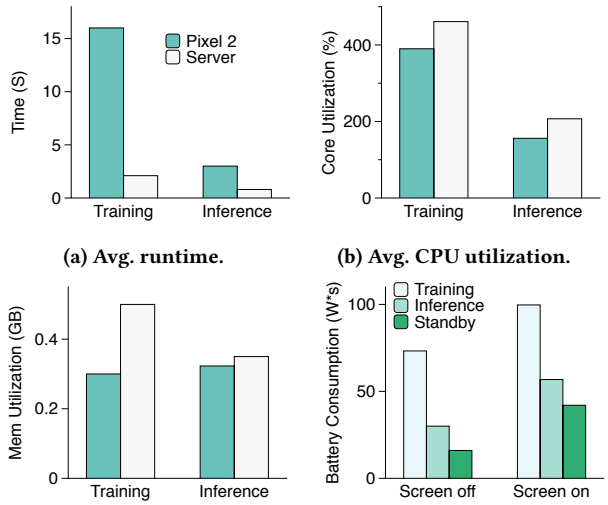
Figure 3d compares the battery consumption of various situations on Pixel 2 while performing deep learning tasks with the Mentee network: (A) training/inference with the device's screen off; (B) training/inference with the device's screen on. We also measured the battery consumption when the device is in standby mode as a baseline. The battery consumption is calculated by multiplying the real-time current and the voltage of the device battery. We measured the real-time current with an interface provided by Android. The commonly used profiler, Trepn [34], is not supported on Pixel 2 but we used it to verify the correctness of our method on the older devices. The results confirm that training has reasonable impact on the device's battery life.

Figure 4 and 5 show the results of the training time and inference time using the reduced, six-layer MobileNet model and the reduced, eight-layer ResNet model with a batch size of 128. The results from training are for only a *single* iteration of one batch of images. From the MobileNet and ResNet results, we can draw similar conclusions as the above Mentee network experiments. For the sake of conciseness, in the rest of the paper, we present only the results from Pixel 2, a more recent generation of mobile devices.

While the above results confirm the feasibility of learning on the mobile devices, they also show that relying solely on CPU is not sufficient for efficient training on the mobile devices. Even though these models are specially designed for mobile devices, the training time is more than 10 seconds for one iteration on a single batch of images. To reach a good accuracy, the training process will take about 100 epochs, which is more than 100 hours on mobile devices. In order to speed up such a lengthy training process, we need to leverage the hardware accelerators (such as GPUs) on mobile devices. We will evaluate TensorFlow+'s support of hardware acceleration in Section 4.5.

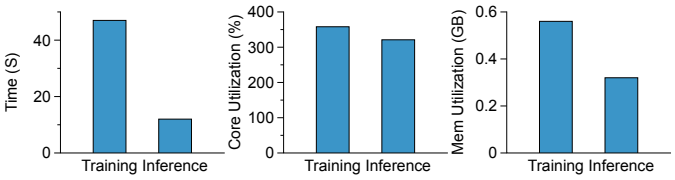
4.3 Fully-connected-layer- and Convolutional-layer Only Models

To further investigate the cost of training DNNs on mobile devices, we took a close look at the most computationally expensive building blocks in DNNs—fully-connected layers and convolutional layers



(a) Avg. runtime. (b) Avg. CPU utilization. (c) Avg. memory utilization. (d) Battery consumption.

Figure 4: Performance and resource usages of the reduced Mobilenet.



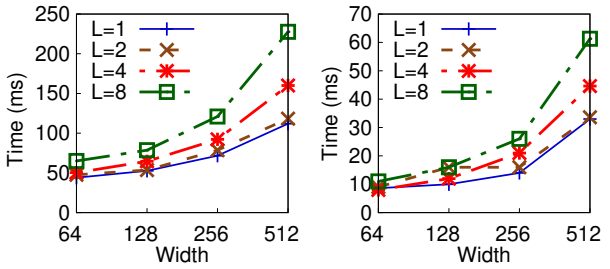
(a) Avg. runtime. (b) Avg. CPU utilization. (c) Avg. memory utilization.

Figure 5: Performance and resource usages of the reduced ResNet.

Table 4: Estimation of the numbers of floating point operations (flops) and parameters in the fully-connected-layer-only models. The first row denotes the width and the first column denotes the depth of a model. The values in each cell are the number of flops and the number of parameters.

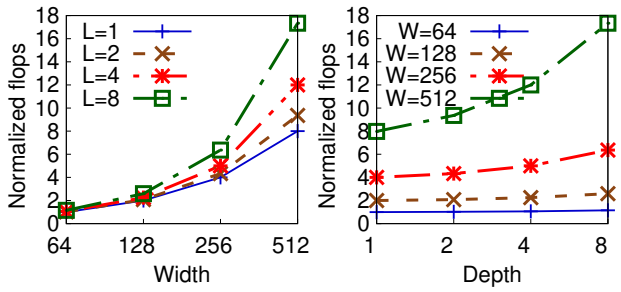
	64	128	256	512
1	15.1G, 196K	30G, 393K	60G, 786K	121G, 1.57M
2	15.4G, 200K	31G, 409K	65G, 852.48K	141G, 1.84M
4	16G, 209K	34G, 442K	75G, 984K	181G, 2.36M
8	17G, 225K	39G, 508K	95G, 1.25M	252G, 3.41M

in the next few subsections. We designed experiments using simple models with only fully-connected layers or only convolutional layers (no activation layer or pooling layer). As the complexity of a network depends on its depth (the number of layers) and width (the number of neurons in each layer), we varied the depth and width of a network to study their impact on the performance. We used CIFAR-10 dataset as input for our tests and the batch size is



(a) Training time. (b) Inference time.

Figure 6: Performance of fully-connected-layer only model.



(a) Flops vs. width. (b) Flops vs. depth.

Figure 7: Normalized flops of a fully-connected-layer-only model with various width and depth.

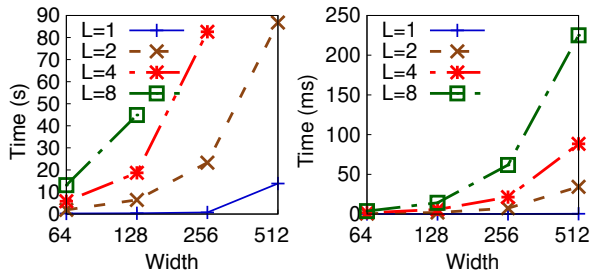
128 in all the following tests. We performed training and inference for a single iteration with one batch of images.

4.3.1 Fully-connected-layer-only Models. Figure 6 demonstrates how the training and inference time vary with different combinations of network depth and width using fully-connected-layer-only models (the numbers of parameters and floating point operations of each model are listed in Table 4). The training time increases by 100% when the depth grows from one layer to eight layers, and it increases by 254% when the width grows from 64 to 512. The inference time increases by up to 85.7% with the increase of depth and it increases by up to 457% with the increase of width. These results show that the training and inference time of fully-connected layers are more sensitive to *width* than to depth.

To confirm the above understanding, we show how the computational cost of a fully-connected-layer-only model changes with respect to the width and depth of the model. We measured the cost in terms of the number of floating point operations (flops) of our model. Figure 7 shows the results normalized to the one-layer model with a width of 64. The results confirm that the computational cost of a fully-connected-layer only model is indeed more sensitive to its width than depth. Every neuron from a fully-connected layer needs to interact will all the neurons from the subsequent layer, which explains its sensitivity toward width. Based on this observation, we believe that narrow but deep models are more suited for mobile devices.

Table 5: Estimation of the numbers of floating point operations (flops) and parameters in the convolutional-layer-only models. The first row denotes the width and the first column denotes the depth of a model. The values in each cell are the number of flops and the number of parameters.

	64	128	256	512
1	1.3G, 1.7K	2.7G, 3.5K	5.5G, 7K	11G, 14K
2	30G, 38.7K	118G, 15K	469G, 597K	618G, 237M
4	88G,112K	350G, 446K	1397G, 1.78M	5577G, 7.09M
8	204G, 260K	814G, 1M	3253G, 4M	13000G, 537M



(a) Training time (b) Inference time

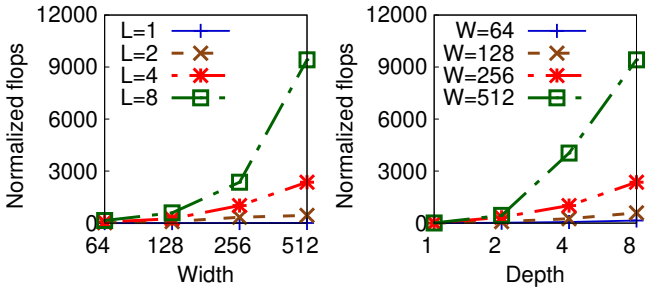
Figure 8: Performance of convolutional-layer-only model.

4.3.2 Convolutional-layer-only Models. Figure 8 shows the training and inference time with various configurations using convolutional-layer-only models (the numbers of parameters and floating point operations of each model are listed in Table 5). The training time increases by 125× when the depth grows from one layer to eight layers, and it increases by 64× when the width grows from 64 to 512. The inference time increases by up to 625× with the increase of the depth and by up to 75× with the increase of width. Hence, the training time of the convolutional-layer-only models is a lot more sensitive to the growth in *depth* than to width, which is in contrast to the observation from fully-connected-layer-only models.

Theoretically, a convolutional layer has a computational cost proportional to: $D_k \times D_k \times M \times N \times D_f \times D_f$ [20], where D_k is the size of the kernel, M and N are the numbers of input and output channels, and D_f is the size of the filter. As a result, doubling the size of M adds about the same cost as introducing a new layer.

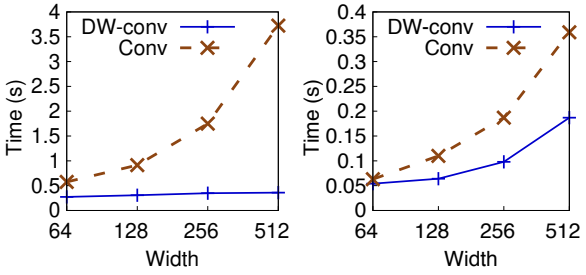
The measurements of flops in training with various widths and depths in the convolutional-layer-only models also confirm the above theory. Figure 9 shows that the number of flops increases much faster when the depth increases compared to when the width increases. The sensitivity of convolutional layers to width also manifests in their memory requirement. Four-layered and eight-layered convolutional-layer-only models cannot even run on the device due to the out of memory error when the network model has a width of 512.

Finally, comparing the training time between models with only fully-connected layers and only convolutional layers, the latter



(a) Flops vs. width. (b) Flops vs. depth.

Figure 9: Normalized flops of a convolutional-layer-only model with various width and depth.

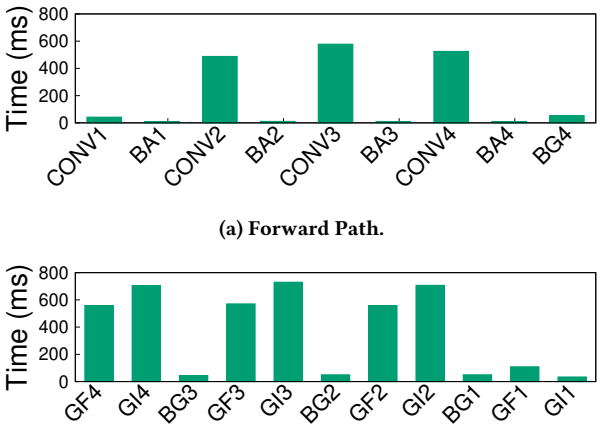


(a) Training time. (b) Inference time.

Figure 10: Performance of depthwise-separable-convolution-layer-only models (DW-conv) and standard convolutional-layer-only models (Conv).

takes on average, two orders of magnitudes more time than the former with the same depth and width. Convolutional-layer-only models have less number of parameters compared to fully-connected-layer-only models when the model is not complex. But the growth of the number of parameters in convolutional-layer-only models is much faster than fully-connected-layers-only models as the model becomes more complex, as shown in Table 4 and 5. The number of parameters of a convolutional-layer only model is about the same as the fully-connected-layer-only model when the width is 128 and the depth is 4. When the model has 8 layers in depth and 512 neurons in width, the number of parameters of the convolutional-layer-only model is 4.85 times more than that of the fully-connected-layer model. The faster growth of the number of parameters also explains why convolutional-layer-only models are more sensitive to the increase of depth than fully-connected-layer-only models.

In order to reduce the computational cost of convolutional layers, Howard et al. proposed depthwise separable convolution [20], which splits the calculation of a standard convolution into two steps: 1) calculate convolution along the depth dimension using multiple kernels and each of the kernels iterates one channel of the input, and 2) merge all the results from the previous step using standard convolution with a one by one kernel. The result shows that depthwise separable convolution uses between 8 to 9 times less computation than the standard convolution, and may be more appropriate for deep learning on mobile devices. Hence, we also



(a) Forward Path.

(b) Backward Path.

Figure 11: Runtime of various operations involved in training a four-layer convolutional-layer-only model (CONV: convolution; BA: bias addition; BG: gradient for bias addition; GF: convolution gradient with respect to the filter; GI: convolution gradient with respect to the input. The number after the operation name is the layer number).

evaluated the performance of depthwise separable convolutional-layer only models.

Figure 10 compares the training time of a single depthwise separable convolution layer (*DW-conv*) and a single standard convolutional layer (*Conv*). With a width of 64, the training time of depthwise convolution is already faster than that of the standard convolution; and it does not grow much as the width increases. When the width is 512, the depthwise separable convolution needs only one third of the training time of the standard convolution. For inference, the speedup of depthwise separable convolution is 2× when the width is 512. The results confirm that depthwise separable convolution indeed has a significant advantage over standard convolution in computational cost.

The main trade-off from using depthwise separable convolution vs. standard convolution is speed vs. accuracy. With the full MobileNet model, our experimental result shows that the accuracy loss of depthwise convolution compared to the standard convolution is 6%, whereas with the reduced MobileNet model, the accuracy loss is 12%. In comparison, the speedup that we observed from using depthwise separable convolution is 3×, which makes it a worthy option for deep learning on resource-constrained devices.

4.4 Training Time Breakdown

The previous results show that training is much more expensive than inference, especially for the models with convolutional layers, on mobile devices. As illustrated in Figure 1, the training process involves more computational intensive operations, such as gradient calculation. In order to understand which operations contribute to the training time and how we can possibly improve it on mobile devices, we analyzed the performance of the various operations involved in the training process.

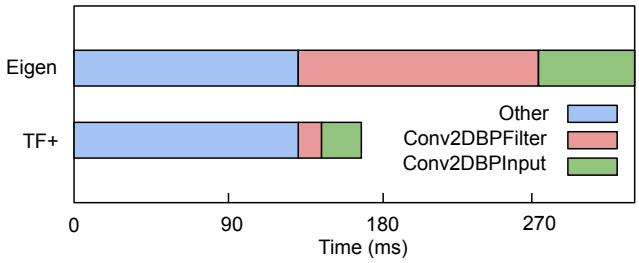
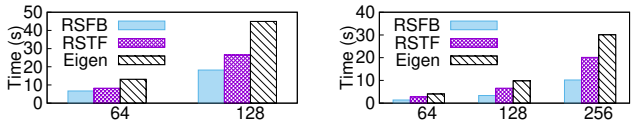


Figure 12: Accelerating various operations involved in training a single convolutional layer using GPU. The calculation time of *Conv2DbackpropInput* is reduced by 37.5%, and that of the *Conv2DBackpropFilter* is reduced by 90.6%.



(a) Eight-layered model. **(b) Four-layered model.**

Figure 13: Training time of convolutional-layer-only models with acceleration in both forward and backward path (TF+), acceleration only in forward path (RSTF), and no acceleration (Eigen). The convolutional-layer-only models use various width, ranging from 64 to 256.

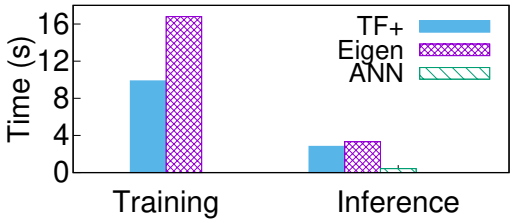


Figure 14: Performance of the reduced MobileNet running on GPU (TF+), CPU (Eigen), and specialized accelerator (ANN).

Figure 11 illustrates the training time breakdown of a four-layer convolutional-layer-only model. The convolution calculation time is proportional to the input size in both the forward path and the backward path. We also observe that in the forward path, the convolution operations are the slowest whereas in the backward path, the convolution gradient calculation is the most time-consuming. In particular, gradient calculation time accounts for 91% of the backward path calculation time and 65% of the total computation latency. The above result motivates us to focus our effort on reducing the computation latency of the gradient calculation so that mobile devices can support deep learning tasks more efficiently.

4.5 Hardware Acceleration

In this section, we evaluate the performance improvement made by Tensorflow+ from using on-device GPU to accelerate training. Following the discussion in the previous section, we first evaluate the effectiveness of accelerating the various components involved in training a single convolutional layer. Figure 12 shows that TensorFlow+ reduces the calculation time in the gradient calculation compared to TensorFlow's CPU implementation. In particular, the calculation time of *Conv2DBackpropInput* is reduced by 37.5%, and that of the *Conv2DBackpropFilter* is reduced by 90.6%. The *other* operations are not accelerated and their runtimes remain unchanged. With this understanding, next we evaluate the speedup achieved by accelerating complete networks.

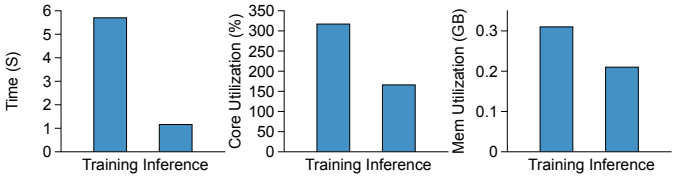
Figure 13 shows the training time of eight-layer and four-layer convolutional-layer-only models with acceleration (*TF+*) and without it (*Eigen*). The results show that hardware acceleration achieves a speedup of up to 2.2×. To show the importance of accelerating the backward path, which is not supported by the previous work [2], we also compared to the performance of using RenderScript to accelerate only the forward path (*RSTF*), which shows a speedup of only 30%. Then, we further evaluated the effectiveness of acceleration on a real network designed for mobile devices, MobileNet. Figure 14 shows the training and inference times for the reduced, six-layer MobileNet with acceleration (*TF+*) and without it (*Eigen*). The results confirm the effectiveness of using hardware acceleration for learning on mobile devices: it achieves a speedup of 1.7× on training and 1.2× on inference.

In addition, we also explore the effectiveness of using the new Pixel Visual Core (PVC) and Android Neural Network API (*ANN*) to accelerate inference, which shows an impressive 5.7× speedup compared to our GPU-based acceleration of inference. The reason for the performance difference between GPU and PVC is two-fold. First, PVC can deliver higher throughput compared to GPU due to its larger numbers of arithmetic logic units (ALUs). PVC consists of eight customized cores and each core has 512 ALUs whereas Adreno 540 has only 256 ALUs. PVC can deliver 3.28 Tflops raw computing power whereas Adreno 540 can achieve only 567 GFlops [14, 35]. Second, PVC uses TensorFlow Lite models, which are less expensive than the TensorFlow models used by GPU. TensorFlow Lite uses an optimizing converter to convert a pre-trained TensorFlow model to a TensorFlow Lite model. The conversion quantizes the weights and biases in a model to reduce the computation and memory footprint; it also fuses activations to provide a better data level parallelism. However, the PVC core supports only inference. Therefore, it well complements the general-purpose GPU, and they can be utilized to accelerate inference and training, respectively, on devices.

4.6 Deep Learning on IoTs

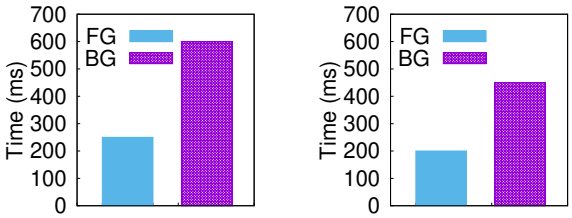
Next, we investigate the capability of IoT devices in performing both training and inference. We evaluate the training and inference on Raspberry Pi 3B+, a commonly used platform for developing IoT applications, using the reduced MobileNet model.

Figure 15 illustrates the training time and inference time running the reduced MobileNet model on Raspberry Pi. The Pi has even less memory than the mobile devices, and we can run a batch size of only four on the reduced MobileNet model. The resource utilization follows a similar pattern as our results on mobile devices. The training time of a single iteration on a batch of images is 6X longer



(a) Avg. runtime (b) Avg. CPU utilization (c) Avg. memory utilization.

Figure 15: Resource utilization of the reduced MobileNet on Raspberry Pi 3B+.



(a) TensorFlow. (b) TF+.

Figure 16: Performance of training in foreground and in background with CPU (TensorFlow) and GPU (TF+).

than the inference time. Both are much slower than on mobile devices. The lack of accelerators makes it even more difficult for IoTs to support deep learning tasks.

4.7 User Experience

4.7.1 Foreground training and background training. Since training is an intensive task, running it in background can reduce its impact on user experience of other applications on the device. Because Android always sets higher priority to the foreground applications compared to the background applications to provide prompt response to user input, more resources are allocated to the foreground applications. Training is inevitably slower in background with less resources compared to foreground. We compared the performance difference between training in foreground and in background on both CPU and GPU in Figure 16, and the results show that the former is 3X faster than the latter.

4.7.2 Impact on Other Applications. We evaluated how learning impacts the performance of other applications running on the same device. We used a mobile benchmark [33], PassMark, to model typical applications on a device. It includes multiple tests which stress different hardware components of the device. The CPU tests in the benchmark, including finding prime numbers, encryption, and compression can model computational intensive tasks. The memory tests measure the access latency of the memory system with varied data size block, and can model the memory intensive tasks. The graphics tests measures the rendering capability of the GPU by rendering images and game-like scenes on the screen.

By comparing the performance from the benchmark with and without running the learning tasks, we can quantitatively evaluate the impact of the learning tasks on other applications. Since

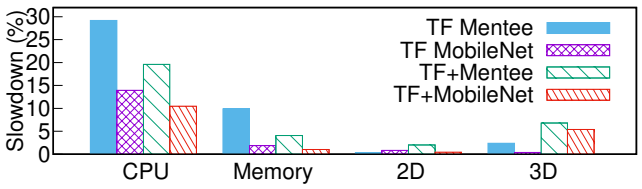


Figure 17: PassMark slowdown with training in background. We compared the PassMark score with various configurations. All the scores are normalized to the baseline PassMark score, which is from running PassMark standalone. *TF Mentee* and *TF MobileNet* run the deep learning tasks on CPU in background along with PassMark in foreground. The *TF+Mentee* and *TF+MobileNet* run deep learning tasks on GPU in background along with PassMark in foreground.

training a model is often time consuming and does not require user interactions, we ran training using TensorFlow+ in background while running the benchmark in foreground. We considered four different cases in which we train the Mentee and MobileNet models with and without GPU. Figure 17 shows the slowdown of PassMark with respect to its standalone performance with no training in background.

The results show that training has a significant performance impact on the system, especially when running on CPU. The CPU utilization is about 305% and 348% for training MobileNet and Mentee, respectively, which leads to up to 30% drop in PassMark's CPU test performance. The results also show that the reduced MobileNet has 15% less slowdown compared to Mentee in the CPU test performance. When the models are run on GPU, we observe that the CPU load is reduced, leading 10% less performance slowdown.

The memory test result shows that training also has a quite significant impact on memory-intensive applications. Training the reduced MobileNet has a less impact than Mentee, because the former's network is relatively smaller. Since CPU and GPU share memory on mobile SoCs, offloading training to GPU cannot reduce the impact on memory performance.

In terms of graphic experience, the scores in both 2D and 3D rendering drop about 5% while the mobile device is running the training tasks. It is worth noting that the Mentee network is able to utilize more GPU (50% more) than MobileNet because of its wider network which increases data parallelism. But, overall, the performance impact of training on a GPU-intensive application is much less significant than on a CPU-intensive application, which tells us that the mobile device's GPU is quite capable.

4.7.3 Impact on User Interactions. We also investigated the impact on user interactions when running deep learning tasks. Our goal is to find out whether running deep learning tasks affects a user's interactive experience with the device. We performed our measurements using an application which models user interaction with a device by taking user input from the touch screen and then performing various tasks. Based on the user input, the mobile device will render different output responses on the screen. By analyzing the frame rendering data in response to the user input, we

Table 6: Impact of background training on user interactions. The response time shows how fast the application react to user input events. The frame latency is the time to render one frame upon a user input event.

	Baseline	CPU	GPU
99th percentile response time (ms)	0.21	0.24	0.21
99th percentile frame latency (ms)	5.49	5.75	5.78

can understand the user interactivity quantitatively. We used the Android *dumpsys* tool [8] to collect aggregated analysis of frame rendering data. There are two metrics of interest: 1) response time, the time for the application to process user inputs; and 2) frame latency, the time to complete rendering a new frame based on the user inputs. We again stressed the system by training the Mentee network, which is more computational intensive compared to the reduced MobileNet, on Pixel 2.

Table 6 lists the 90th percentile response time and frame latency of three different settings, 1) without training in background (*baseline*), 2) training the Mentee network on CPU (*CPU*), and 3) training the Mentee network on GPU (*GPU*). The results show that the deep learning tasks have minimal impact on the processing of user inputs. The response time increases by merely 0.03 ms when the training is running on CPU, and it remains the same when the training is on GPU. We can conclude that the impact from the deep learning tasks is not perceivable by users, because the minimum threshold of perceivable latency is 100 ms [4, 31, 32].

The frame latency results show that the application can render more than 60 frames per second when training either on CPU or GPU. The frame latency increases only 0.26 ms and 0.29 ms, when training on CPU and GPU, respectively. If an application is rendering at 60 frames per second, the deadline for rendering each frame is 16 ms (1000ms/60 = 16ms). The additional frame latency incured by running deep learning tasks is negligible, which means that the background training is not affecting the graphic experience provided by the application.

5 CONCLUSIONS

This paper presents a comprehensive study on the software and hardware capability of mobile devices in supporting deep learning tasks. We conclude that mobile devices can support both inference and training using DNNs with reasonable performance and reasonable impact on user experience. But due to the limited resource availability on mobile devices, we need to carefully design our deep learning models to control the complexity, especially for training. New learning paradigms such as federated learning and knowledge transfer are promising approaches to utilizing mobile devices and improving deep learning. Our work sheds light on the effectiveness of such approaches on modern smartphones and IoT platforms.

Specifically, our findings show that the width and depth of a network have different performance impact on different types of layers. Fully-connected layers are more sensitive to width whereas convolutional layers are more sensitive to depth. This insight can help us design models that are suitable for mobile devices. Our findings also show that utilizing GPUs on mobile devices is important to the performance of training on devices, whereas specialized image and AI processors can achieve substantial speedup for inference. Hence, combining the use of these accelerators can be of great benefit to deep learning tasks on mobile devices. Another insight is that running the deep learning tasks in background on a device has a minor impact on the foreground tasks and user interactions as well as the resource and battery usages.

Our solution, TensorFlow+, embodies several key extensions to TensorFlow for deep learning on mobile devices, including the support of training and GPU-based acceleration. It is open source and publicly accessible [27].

6 ACKNOWLEDGMENT

This research is sponsored by U.S. National Science Foundation CAREER award CNS-1619653 and awards CNS-1562837, CNS-1629888, IIS-1633381, and CMMI-1610282. We thank our shepherd, Aakanksha Chowdhery, and the anonymous reviewers for their helpful suggestions. We would also like to thank Kaiqi Zhao, Sungho Hong, and Eugene Kuznetsov for their help on reviewing the paper.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Moustafa Alzantot, Yingnan Wang, Zhengshuang Ren, and Mani B Srivastava. 2017. Rstensorflow: GPU enabled tensorflow for deep learning on commodity android devices. In *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*. ACM, 7–12.

[3] Jimmy Ba and Rich Caruana. 2014. Do deep nets really need to be deep?. In *Proceedings of Advances in neural information processing systems*. 2654–2662.

[4] Stuart K Card, George G Robertson, and Jock D Mackinlay. 1991. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*. ACM, 181–186.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[6] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *Proceedings of International Conference on Machine Learning*. 2285–2294.

[7] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. 2014. DeCAF: A deep convolutional activation feature for generic visual recognition. In *Proceedings of International conference on machine learning*. 647–655.

[8] Dumpsys. 2015. Dumpsys - A tool to provide information about system services on Android devices. https://developer.android.com/studio/command-line/dumpsys.

[9] Eigen. 2017. Eigen Library. http://eigen.tuxfamily.org.

[10] Facebook. 2017. Caffe2 - A new lightweight, modular, and scalable deep learning framework. https://caffe2.ai/.

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of Advances in neural information processing systems*. 2672–2680.

[12] Google. 2013. RenderScript Overview. https://developer.android.com/guide/topics/renderscript/compute.

[13] Google. 2017. Introduction to TensorFlow Lite. https://www.tensorflow.org/mobile/tflite.

[14] Google. 2017. Pixel Visual Core (PVC) - Google. https://en.wikichip.org/wiki/google/pixel_visual_core. IPU.

[15] Google. 2017. TensorFlow Mobile. https://www.tensorflow.org/mobile/android_build.

[16] Google. 2019. On-Device Training with TensorFlow Lite. https://github.com/tensorflow/community/pull/124.

[17] Hervé Guihot. 2012. RenderScript. In *Pro Android Apps Performance Optimization*. Springer, 231–263.
[18] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[20] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[21] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. 2018. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2752–2761.
[22] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient data encoding for deep neural network training. In *Proceedings of 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 776–789.
[23] Deepak Kadetotad, Sairam Arunachalam, Chaitali Chakrabarti, and Jae-sun Seo. 2016. Efficient memory compression in deep neural networks using coarse-grain sparsification for speech applications. In *Proceedings of Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 1–8.
[24] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 615–629.
[25] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).
[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of Advances in neural information processing systems*. 1097–1105.
[27] ASU VISA Research Lab. 2019. TensorFlow+: A GPU accelerated deep learning framework for on-device training. https://github.com/ychen404/TensorFlowPlus.
[28] Nicholas D Lane and Petko Georgiev. 2015. Can deep learning revolutionize mobile sensing?. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 117–122.
[29] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Proceedings of Advances in neural information processing systems*. 598–605.
[30] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 19–34.
[31] Robert B Miller. 1968. Response time in man-computer conversational transactions. In *Proceedings of AFIPS Fall Joint Computing Conference (1)*. 267–277.
[32] Brad A Myers. 1985. The importance of percent-done progress indicators for computer-human interfaces. In *ACM SIGCHI Bulletin*, Vol. 16. ACM, 11–17.
[33] PassMark. 2015. PassMark Software - PerformanceTest System Benchmarks. http://www.passmark.com/baselines/index.php. PassMark.
[34] Qualcomm. 2014. Trepn profiler. developer.qualcomm.com. Qualcomm Inc. Trepn profiler.
[35] Qualcomm. 2017. Adreno. https://en.wikipedia.org/wiki/Adreno. Adreno.
[36] S Rallapalli, H Qiu, A Bency, S Karthikeyan, R Govindan, B Manjunath, and R Urgaonkar. 2016. Are very deep neural networks feasible on mobile devices. *IEEE Trans. Circ. Syst. Video Technol* (2016).
[37] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. 1988. *The C programming language*. Prentice Hall Englewood Cliffs.
[38] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. FitNets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550* (2014).
[39] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.
[40] Ragini Sharma, Saman Biookaghazadeh, Baoxin Li, and Ming Zhao. 2018. Are existing knowledge transfer techniques effective for deep learning with edge devices?. In *Proceedings of 2018 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 42–49.
[41] Laurent Sifre and PS Mallat. 2014. *Rigid-motion scattering for image classification*. Ph.D. Dissertation.
[42] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. 2017. Federated multi-task learning. In *Proceedings of Advances in Neural Information Processing Systems*. 4424–4434.
[43] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.
[44] Ragav Venkatesan and Baoxin Li. 2016. Diving deeper into mentee networks. *arXiv preprint arXiv:1604.08220* (2016).
[45] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).
[46] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. 2018. Deep Learning in Mobile and Wireless Networking: A Survey. *arXiv preprint arXiv:1803.04311* (2018).