A First Look at Deep Learning Apps on Smartphones

Mengwei Xu
Key Lab of High-Confidence Software Technologies (Peking University), MoE, Beijing, China
mwx@pku.edu.cn

Jiawei Liu
Key Lab of High-Confidence Software Technologies (Peking University), MoE, Beijing, China
jiaweiliu@pku.edu.cn

Yuanqiang Liu
Key Lab of High-Confidence Software Technologies (Peking University), MoE, Beijing, China
yuanqiangliu@pku.edu.cn

Felix Xiaozhu Lin
Purdue ECE
West Lafayette, Indiana, USA
xzl@purdue.edu

Yunxin Liu
Microsoft Research
Beijing, China
yunxin.liu@microsoft.com

Xuanzhe Liu
Key Lab of High-Confidence Software Technologies (Peking University), MoE, Beijing, China
xzl@pku.edu.cn

ABSTRACT

To bridge the knowledge gap between research and practice, we present the first empirical study on 16,500 the most popular Android apps, demystifying how smartphone apps exploit deep learning in the wild. To this end, we build a new static tool that dissects apps and analyzes their deep learning functions. Our study answers threefold questions: what are the early adopter apps of deep learning, what do they use deep learning for, and how do their deep learning models look like. Our study has strong implications for app developers, smartphone vendors, and deep learning R&D. On one hand, our findings paint a promising picture of deep learning for smartphones, showing the prosperity of mobile deep learning frameworks as well as the prosperity of apps building their cores atop deep learning. On the other hand, our findings urge optimizations on deep learning models deployed on smartphones, protection of these models, and validation of research ideas on these models.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Human-centered computing → Ubiquitous and mobile computing**.

KEYWORDS

Empirical Study; Deep Learning; Mobile Apps

ACM Reference Format:
Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558.3313591

1 INTRODUCTION

Smartphones are undoubtedly among the most promising platforms for running deep learning (DL) based applications [3, 6, 8, 27]. Such a huge market is driven by continuous advances in DL, including the introduction of latest neural-network (NN) hardware [45, 47, 57, 103], improvements in DL algorithms [53, 77, 82, 89], and the increased penetration in huge information analytics [49, 54, 79, 81]. The research community has built numerous DL-based novel apps [44, 68, 69, 76, 83, 102]. The industry has also tried to utilize DL in their mobile products. For example, in the latest released Android 9 Pie OS, Google introduces a small feed-forward NN model to enable Smart Linkify, a useful API that adds clickable links when certain types of entities are detected in text [38].

The year 2017 marked the dawn of DL for smartphones. Almost simultaneously, most major vendors roll out their DL frameworks for smartphones, or *mobile DL framework* for short. These frameworks include TensorFlow Lite (TFLite) from Google [37] (Nov. 2017), Caffe2 from Facebook [9] (Apr. 2017), Core ML from Apple [12] (Jun. 2017), ncnn from Tencent [35] (Jul. 2017), and MDL from Baidu [24] (Sep. 2017). These frameworks share the same goal: executing the DL inference task solely on smartphones. Compared to offloading the DL inference from smartphones to the cloud [2, 19, 21], the on-device DL inference better protects user privacy without uploading sensitive data, continues to operate in the face of poor Internet connectivity, and relieves app authors from affording the cost of running DL in the cloud [45, 63, 66, 67, 67, 69, 74, 100, 103].

Following the DL framework explosion, there emerges the first wave of smartphone apps that embrace DL techniques. We deem it crucial to understand these apps and in particular how they use DL, because the history has demonstrated that such *early adopters* heavily influence or even decide the evolution of new technologies [86] – smartphone DL in our case.

To this end, we present the first empirical study on how real-world Android apps exploit DL techniques. Our study seeks to answer *threefold* questions: *what are the characteristics of apps that have adopted DL, what do they use DL for*, and *what are their DL models.* Essentially, our study aims findings on how DL is being used by smartphone apps *in the wild* and the entailed implications, filling a key gap between mobile DL research and practice.

For the study, we have examined an extensive set of Android apps from the official Google Play appstore. We take two snapshots of the app market in early Jun. 2018 and early Sep. 2018 (3 months apart), respectively. Each snapshot consists of 16,500 the most popular apps covering 33 different categories listed on Google Play. We derive

Xuanzhe Liu is the paper's corresponding author.

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313591

insights by inspecting individual apps as well as by comparing the two snapshots. To automate the analysis of numerous Android apps, we build a new analyzer tool that inspects app installation packages, identifies the apps that use DL (dubbed "DL apps"), and extracts DL models from these apps for inspection. To realize such a tool, we eschew from looking for specific code pattern and instead identify the usage of known DL frameworks, based on a rationale that most DL apps are developed atop DL frameworks.

Our key findings are summarized as follows.

Early adopters are top apps (§4.2) We have found 211 DL apps in the set of apps collected in Sep. 2018. Only 1.3% of all the apps, these DL apps collectively contribute 11.9% of total downloads of all the apps and 10.5% of total reviews. In the month of Sep. 2018, the 211 DL apps are downloaded for around 13,000,000 times and receive 9,600,000 reviews. DL apps grow fast, indicating a 27% increase in their numbers over the 3 months in our study.

DL is used as core building blocks (§4.3) We find that 81% DL apps use DL to support their core functionalities. That is, these apps would fail to operate without their use of DL. The number of such DL apps grow by 23% over the period of 3 months.

Photo beauty is the top use (§4.3) DL is known for its diverse applications, as confirmed by the usage discovered by us, e.g. emoji prediction and speech recognition. Among them, photo beauty is the most popular use case: 94 (44.5%) DL apps use DL for photo beauty; 61 (29%) DL apps come from the photography category.

Mobile DL frameworks are gaining traction (§4.4) While full-fledged DL frameworks such as TensorFlow are still popular among DL apps due to their momentum, DL frameworks designed and optimized for constrained resources are increasingly popular. For instance, the number of DL apps using TFLite has grown by 258% over the period of 3 months.

Most DL models miss obvious optimizations (§5.2) Despite well-known optimizations, e.g. quantization which can reduce DL cost by up to two orders of magnitude with little accuracy loss [58], we find only 6% of DL models coming with such optimizations.

On-device DL is lighter than one may expect (§5.3) Despite the common belief that the power of DL models comes from rich parameters and deep layers, we find that DL models used in apps are rather small, with the median memory usage of 2.47 MB and the inference computation of 10M FLOPs, which typically incurs the inference delay of tens of milliseconds. These models are not only much lighter than full models designed for cloud/servers (e.g. ResNet-50 with 200 MB memory and 4G FLOPs inference computations) but also lighter than well-known models specifically crafted for smartphones, e.g. MobileNet with 54 MB memory and 500M FLOPs inference computations.

DL models are poorly protected (§5.4) We find that only 39.2% discovered models are obfuscated and 19.2% models are encrypted. The remaining models are a bit trivial to extract and therefore subject to unauthorized reuse.

Summary of implications: Overall, our findings paint a promising picture of DL on smartphones, motivating future research and development. Specifically, the findings show strong implications for multiple stakeholders of the mobile DL ecosystem. **To app developers:** our findings show that DL can be sufficiently affordable on smartphones; developers, especially individuals or small companies, should have more confidence in deploying DL in their apps; interested developers should consider building DL capability atop mobile DL frameworks; a few app categories, notably photography, are most likely to benefit from DL techniques. **To DL framework developers:** our findings encourage continuous development of frameworks optimized for smartphones; our findings also show the urgent need for model protection as the first-class concern of frameworks. **To hardware designers:** our findings motivate DL accelerator designs to give priority to the layers popular among mobile DL models. **To DL researchers:** our findings suggest that new proposal for optimizing DL inference should be validated on lightweight models that see extensive deployment on smartphones in the wild.

In summary, our contributions are as follows.

- We design and implement a tool for analyzing the DL adoption in Android apps. Capable of identifying the DL usage in Android apps and extracting the corresponding DL models for inspection, our tool enables automatic analysis of numerous apps for DL.
- We carry out the **first** large-scale study of 16,500 Android apps for their DL adoption. Through the empirical analysis, we contribute new findings on the first wave of apps that adopt DL techniques. In the dawn of DL explosion for smartphones, our findings generate valuable implications to key stakeholders of the mobile ecosystem and shed light on the evolution of DL for smartphones. We also publicize our tools and datasets[1].

2 BACKGROUND

DL models and frameworks DL has revolutionized many AI tasks, notably computer vision and natural language processing, through substantial boosts in algorithm accuracy. In practice, DL algorithms are deployed as two primary parts. The first one is *DL models*, which often comprise neuron layers of various types, e.g. convolution layers, pooling layers, and fully-connected layers. Based on the constituting layers and their organizations, DL models fall into different categories, e.g., Convolutional Neural Network (CNN) containing convolution layers, and Recurrent Neural Network (RNN) processing sequential inputs with their recurrent sub-architectures. The second part is *DL frameworks* that produce DL models (i.e. training) and execute the models over input data (i.e. inference). Since a commodity DL framework often entails tremendous engineering efforts, most app developers tend to exploit existing frameworks by major vendors, such as *TensorFlow* from Google.

Deploying mobile DL As training models is intensive in both data and computing [73], smartphone developers often count on cloud servers for modeling training offline prior to app deployment. At app installation time, the trained models are deployed as part of the app installation package. At runtime, apps perform inference

[1]https://github.com/xumengwei/MobileDL

with the trained models by invoking DL frameworks, and therefore execute AI tasks such as face recognition and language translation.

Inference: on-cloud vs. on-device Towards enabling DL on smartphones, model inference can be either offloaded to the cloud or executed solely on smartphones. Offloading to the cloud is a classical use case of Software-as-a-Service (SaaS), and has been well studied in prior work [43, 62, 85, 101]. The mobile devices upload data and retrieve the inference results, transparently leveraging rich data center resources as server-class GPU. Yet, we have observed that the on-device DL inference is quickly gaining large popularity due to its unique advantages of stronger privacy protection, resilience against poor Internet connectivity, and lower cloud cost to app developers. We will present more evidence in the paper. In this work, we focus our empirical study on such on-device deep learning for smartphones.

3 GOAL AND METHODOLOGY

3.1 Research Goal

The goal of our study is to demystify how smartphone apps exploit DL techniques in the wild. Our study focuses on two types of subjects: i) smartphone apps that embrace DL, and ii) the DL frameworks and models used in practice. Accordingly, we characterize the apps, the frameworks, and the models. We will present the results in Section 4 and Section 5 respectively.

Scope We focus our analysis on Android apps, as Android represents a dominant portion of smartphone shipment (88% in the second quarter of 2018) and hence serves a good proxy for the entire smartphone app population [18].

Datasets We retrieve from the Google Play store the full dataset used in this work. We select 16,500 apps in total, which consist of the top 500 free apps with most downloads from each of the 33 categories defined by Google Play[2]. We have crawled two datasets at different moments, June 2018 and September 2018, which are three months apart. The two app datasets have more than 2/3 overlapped apps. For each app, we download its apk file and crawl its meta information (e.g. app description and user rating) from the Google Play web page for analysis. Our analysis primarily focuses on the newer dataset, i.e., Sep. 2018, and the difference between the two data sets, unless specified otherwise.

3.2 Workflow Overview

We design and implement an analyzing tool to enable our research goal on large-scale Android apps. The tool runs in a semiautomatic way, as illustrated in Figure 1.

The very first step of the analyzing tool is identifying DL apps among a given set of Android apps as input. This is achieved via the module named DL Sniffer. The core idea of DL Sniffer, is detecting the usage of popular DL frameworks, instead of directly finding the usage of DL. After identifying DL apps, it performs analysis on those apps. During analysis, we use the manifest files extracted from DL apps via tool aapt [4] and the meta information crawled from the corresponding Google Play web page. The manifest files include information such as package name, app version, required permissions, etc. The web pages include information such as app description, user rating, app developer, etc.

The analyzing tool further extracts DL models from those DL apps. This extraction is achieved via a module called Model Extractor. After extracting DL models, it performs analysis on them. However, we here face the challenge that the models are mostly in different formats. Although developers are investing substantial efforts in integrating different model formats, such as designing a standardized one [25], the ecosystem of DL frameworks is still broken and fragmented nowadays. Thus, when looking into the internal structures of DL models, we substantially leverage the available tools and source of different frameworks. Fortunately, most of the frameworks we investigated (details in Table 2) are open-source and well-documented.

We discuss more details of DL Sniffer and Model Extractor in Section 4.1 and Section 5.1, respectively.

4 APPLICATION ANALYSIS

This section presents our analysis of smartphone DL apps. We first describe our methodology in Section 4.1 and then the following three major aspects of the DL apps:

- The characteristics of DL apps (§4.2): their popularity, their difference from non-DL apps, and their developers.
- The role of DL (§4.3): the popular usage of DL in apps, the categories of DL apps, and evidence that DL is already used as core building blocks of apps.
- An analysis of DL frameworks (§4.4): which frameworks are used, the cost, and their adoption trend over time.

4.1 Methodology: finding DL apps

As a part of our analyzing tool (Section 3.2), DL Sniffer takes apk file(s) as input, and outputs which of them use DL technique. Detecting DL usage is difficult, instead DL Sniffer mainly detects the usage of popular DL frameworks with Android support. Currently, DL Sniffer supports the detection of 16 popular DL frameworks, including *TensorFlow, Caffe, TFLite,* etc, and the details of those frameworks will be presented later in Section 4.4 & Table 2. DL Sniffer uses two ways to mine the usage of DL frameworks: (1) For those who provide native C++ APIs, DL Sniffer first decomposes the apk files via Apktool [5], and extracts the native shared libraries (with suffix ".so"). DL Sniffer then searches for specific strings on the *rodata* section of those libraries. Those strings can be regarded as identifications of corresponding frameworks, and are pre-defined by us. For example, we notice that the shared libraries that use *TensorFlow* always have *"TF_AllocateTensor"* in its *rodata* section. (2) For those who only support Java APIs, DL Sniffer first converts the apk file into smali code via dex2jar [14]. The smali code, which is a disassembled version of the DEX binary used by Android's Davik VM, enables us to carry out static program analysis. DL Sniffer statically goes through the class/method names of smali code and checks whether certain APIs exist. For example, the class *MultiLayerConfiguration* is used in almost every app that embeds DeepLearning4J framework.

Besides detecting the usage of DL frameworks, we also try to identify DL apps that don't use the frameworks listed in Table 2 (called "*no lib*" in this work). Similarly, this is achieved by searching

[2]Different app categories on Google Play can be visited via url https://play.google.com/store/apps/category/XXX, where XXX can be GAME or other category names.

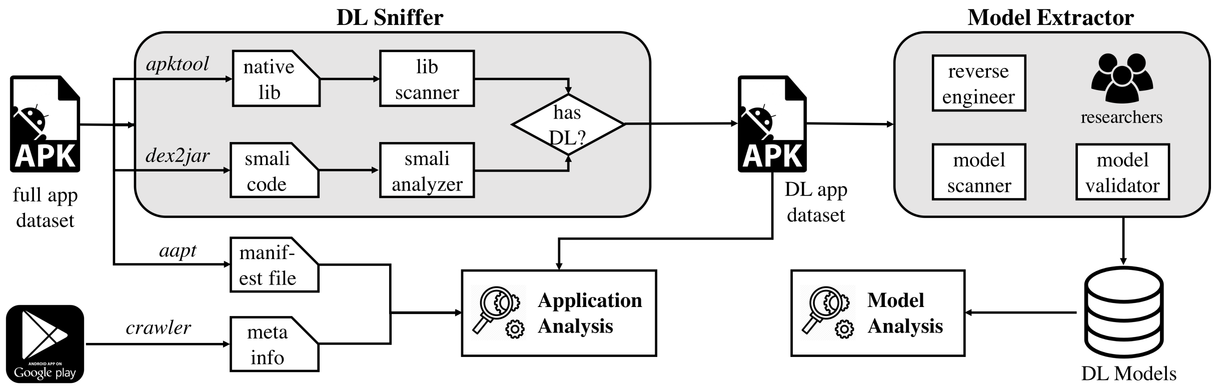

Figure 1: The overall workflow of our analyzing tool.

specific strings on the *rodata* section of native libraries as mentioned above, but the strings we use here are pre-defined such as *"neural network", "convolution", "lstm"*, etc, rather than extracted from existing frameworks. We then manually check the detection correctness through reverse engineering, filtering those don't really have DL usage (false positive). This manual check is also performed on other DL apps detected using the aforementioned approach, to ensure good accuracy in DL identification.

4.2 Characteristics of DL apps

• **Is DL gaining popularity on smartphones?** Our study shows: over our investigation period (June 2018 – Sept. 2018), the total amount of DL apps has increased by 27.1%, from 166 to 211. We further investigate the new downloads and reviews of DL apps within one month of Sept. 2018: in that period, the 211 DL apps are downloaded for around 13,000,000 times and receive 9,600,000 new reviews. The results indicate that a substantial amount of smartphones are running DL apps nowadays.

• **How are DL apps different from non-DL apps?** We investigate the following three aspects with results illustrated in Figure 2.

Downloads and Reviews are typical signal of apps' popularity [75]. As observed, the median number of downloads and reviews of DL apps are 5,000,000 and 41,074 respectively, much larger than non-DL apps, i.e., 100,000 and 1,036 respectively. We also count the download rankings of each DL apps within the corresponding category. The median number of such ranking is 89 among total 500 apps for each category. We deem the above statistics as strong evidences that *top apps are early adopters in deploying DL in mobile apps.* Such phenomenon can be explained that making DL work in the wild, though appealing, takes a lot of engineering efforts. The cycle of developing DL functionality on smartphones includes model construction, data collection, model training, offline/online testing, etc. Thus, many small companies or individual developers lack the resources to exploit DL on their apps.

Ratings show how much appreciation users give to apps. The Figure shows that DL apps and non-DL apps have similar ratings from users, with the same median number 4.3.

App size: as shown in Figure 2, DL apps have much larger apk files than non-DL apps (median number: 35.5MB vs 12.1MB). This

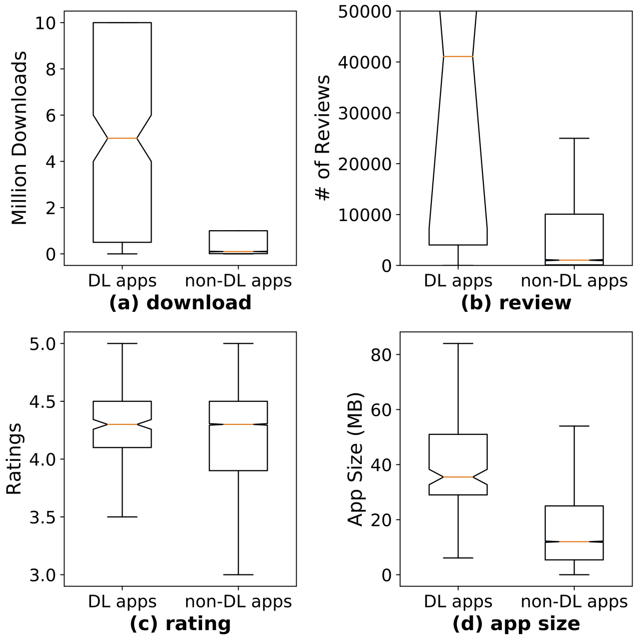

Figure 2: Comparisons between DL apps and non-DL apps on various aspects (a)–(d). Each box shows the 75th percentile, median, and 25th percentile from top to bottom. We manually set the y-axis limits for better presentation in (b): the missed out 75th percentile of DL apps is 324,044.

is reasonable since having DL not only adds DL frameworks and models to the apps, it also confidently indicates that the apps have much richer features.

• **Who are the developers of DL apps?** We also study the developers of DL apps. The results show that the identified 211 DL apps belong to 172 developers (companies), among which 27 developers have more than one DL apps. The developers with most DL apps are "Google LLC" (10) and "Fotoable,Inc" (6). We observe many big companies own more than one DL app, including Google, Adobe, Facebook, Kakao, Meitu, etc. This suggests that those big

Table 1: The DL usage in different apps. Note: as one app may have multiple DL uses, the sum of detailed usage (column 2) might exceed the corresponding coarse usage (column 1).

usage	detailed usage	as core feature
image: 149	photo beauty: 97	94 (96.9%)
	face detection: 52	44 (84.6%)
	augmented reality: 19	5 (26.3%)
	face identification: 8	7 (87.5%)
	image classification: 11	6 (54.5%)
	object recognition: 10	9 (90%)
	text recognition:11	4 (36.3%)
text:26	word&emoji prediction: 15	15 (100%)
	auto-correct: 10	10 (100%)
	translation: 7	3 (42.8%)
	text classification: 4	2 (50%)
	smart reply: 2	0 (0%)
audio: 24	speech recognition: 18	7 (38.9%)
	sound recognition: 8	8 (100%)
other: 19	recommendation: 11	2 (18.1%)
	movement tracking: 9	4 (44.4%)
	simulation: 4	4 (100%)
	abnormal detection: 4	4 (100%)
	video segment: 2	1 (50%)
	action detection: 2	0 (0%)
total: 211		171 (81.0%)

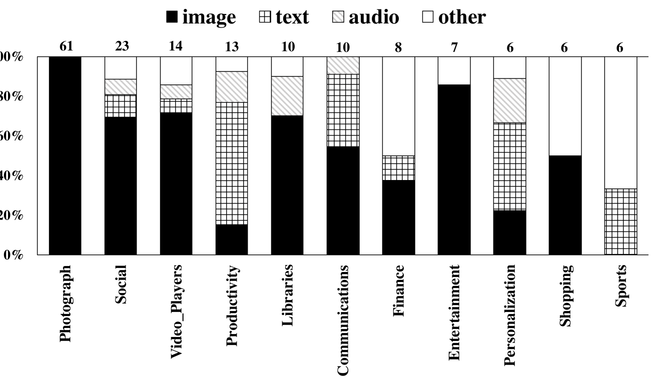

Figure 3: Distributions of DL apps over categories defined by Google Play. Numbers on top: the counts of DL apps in the corresponding categories. Apps in each category are further broken down by DL usage (see Table 1 for description). Categories with fewer than 5 DL apps are not shown.

companies are pioneers in adopting DL into their products. We also notice that DL apps from the same developer often have identical DL frameworks. For example, four products from Fotoable Inc use the exactly same native library called *libncnn_style.0.2.so* to support DL technique. This is because that DL frameworks and even the DL models are easily reusable: a good nature of DL technique that can help reduce the engineering efforts of developers.

Implications: *The popularity of DL among top smartphone apps, especially ones developed by big companies, should endow smaller companies or independent developers with strong confidence in deploying DL in their apps.*

4.3 The roles of DL in apps

• **What are the popular uses of DL?** To understand the roles played by DL, we manually classify the usage of DL on different apps. This is achieved by looking into the app description and app contents. The results are shown in Table 1. Each app has one or more usages, and the usage is represented in two different levels (coarse and detailed). 10 apps are left out since we cannot confirm their DL usage.

Overall, image processing is the most popular usage of DL on smartphones, far more than text and audio processing (149 vs. 26 & 24). This is not surprising since computer vision is the field where DL starts the revolution [64], and the progress on this field has been lasting since then [72]. In more details, photo beauty (97) and face detection (52) are mostly widely used in DL apps, usually found in photo editing and camera apps to beautify pictures. In text field, word & emoji prediction (15) and auto-correct (10) are also popular, usually found in input method apps like GBoard. For audio processing, DL is mainly used for speech recognition (14). Besides, there are other types of usage such as recommendation (11) which is often found in shopping apps.

• **Which categories do DL apps come from?** Figure 3 summarizes the number of DL apps in different categories. As observed, almost 29% DL apps (61 out of 211) are in category photograph, all of which use DL for image processing. Social category is another hotspot with 23 DL apps in total, 78% of which use DL for image processing while others use it for text, audio, etc. The category of productivity also contains 13 DL apps, but most of them (62%) use DL for text processing. Overall, we can see that the DL usage is somehow diverse, with 11 categories has more than 5 DL apps among the top 500. Such diversity gives credits to the good generality of DL technique.

Implications: *Our findings encourage developers of certain types of apps, notably the ones with photo beauty, to embrace DL. Our findings also motivate encapsulating DL algorithms within higher level abstractions that cater to popular uses in apps. For instance, companies such as SenseTime already starts to ship DL-based face detection libraries to app developers. Masking the details of DL models and frameworks, such abstractions would make DL more friendly to developers.*

• **Is DL a core building block?** We also manually tag each DL usage as core feature or not. We define the DL functionality as apps' core feature if and only if two conditions are satisfied: (1) *hot*: the DL functionality is very likely to be invoked every time the apps are opened and used by users, (2) *essential*: without the DL functionality, the apps' main functionality will be severely compromised or even become infeasible. For example, DL on *text recognition* is treated as core feature in a scanner app (Adobe Scan) that helps users translate an image into text, but not in a payment app (Alipay) that uses it to scan ID card for identification. Similarly, DL on *photo beauty* is treated as core feature in a camera app (Meitu), but not in a social app (Facebook Messenger Kids).

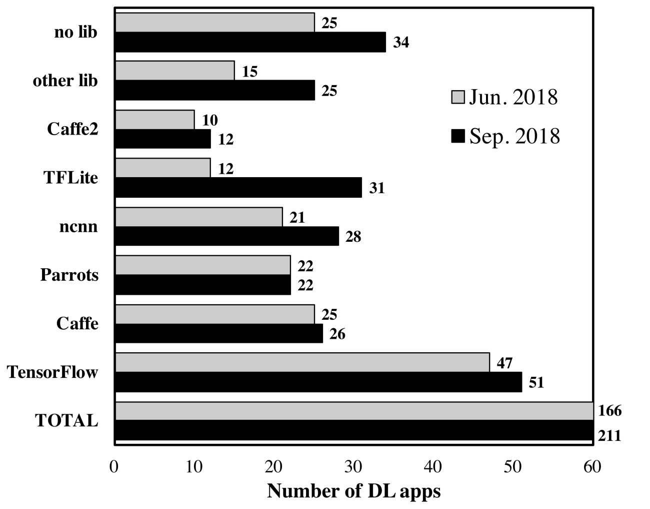

Figure 4: Numbers of DL apps using various mobile DL frameworks. "other lib": the DL apps developed on the frameworks in Table 2 but not itemized here, e.g. *mace, SNPE, and xnn.* "no lib": apps with DL functions but using no DL frameworks from Table 2. Note that the number in "TOTAL" is lower than the sum of others since some DL apps have integrated multiple frameworks.

Overall, 171 out of 211 (81%) apps use DL to support core features. Specifically, since photo beauty (96.9%) and face detection (84.6%) are primarily used in photo & camera apps, their usage is essential. Similarly, word & emoji prediction (100%) and auto-correct (100%) are treated as core features in keyboard apps, helping users input more efficiently and accurately. However, recommendation (18.1%) is often provided as complementary feature to others such as shopping apps, thus not treated as core feature.

Implications: *Our findings support future investment on R&D of mobile DL, as the core user experience on smartphones will probably depend on DL performance [67] and security [80, 92].*

4.4 DL frameworks

As mentioned in Section 2, DL frameworks are critical to DL adoption, as most developers use those frameworks to build their DL apps. In this subsection, we investigate into how those frameworks are used in DL apps: the numbers, the sizes, the practice, etc.

• **A glance over popular DL frameworks** We first make an investigation into popular DL frameworks, and the results are summarized in Table 2. We select those 21 frameworks for their popularity, e.g., forks and stars on GitHub, gained attention on StackOverflow and other Internet channels. Among those 21 frameworks, 16 frameworks support Android platform via Java (official language on Android) and/or C++ (native support via cross-compilation). Most of them are open-source, while others are either provided to public as a binary SDK (*SNPE, CoreML*), or only accessible by the providers' (collaborators') own products (*xNN, Parrots*). Most of them use customized format to store and represent the model files, but some

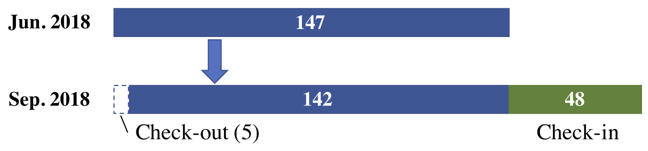

Figure 5: The number of check-in and check-out DL apps.

leverage existing approaches, such as ProtoBuf [28]. We also notice a trend on lightweight DL inference frameworks, which are designed specifically for mobile apps but have no training-support back-end (*ncnn, FeatherCNN, MACE*, etc). Those frameworks cannot train DL models, but can predict with pre-trained models via other frameworks such as *TensorFlow* or *Caffe*. Note that our later analysis substantially relies on the openness of DL frameworks: it enables us to use the existing tools to analyze or even visualize the DL models such as *TensorFlow*, or build our own interpreting scripts to analyze them based on the open code such as *ncnn*.

• **What is the adoption of DL frameworks?** As summarized in Figure 4, the most popular DL frameworks used in Sep. 2018 are *TensorFlow* (51), *TFLite* (31), and *ncnn* (28), as they contribute to almost 50% of the total number of DL apps. Other popular frameworks include *Caffe*, *Parrots*, and *Caffe2*. We have made several observations from those 6 dominant frameworks as following.

(1) All these frameworks are developed by big companies (e.g. Google), AI unicorns (e.g. SenseTime), or renowned universities (e.g. Berkeley).
(2) 5 out of these 6 frameworks are open-source, except *Parrots* which is provided as SDK to consumers. In fact, it is believed that openness is already an important feature in machine learning, especially DL society, as it supposes to [91]. It helps developers reproduce the state-of-the-art scientific algorithms, customize for personal usage, etc. As a result, for example, *TensorFlow* has more than 1,670 contributors up to Oct. 2018, going far beyond the community of Google.
(3) Most (4 out of 6) frameworks are optimized for smartphones, except *Caffe* and *TensorFlow*. Those mobile DL frameworks are designed and developed specifically for mobile devices, usually without training back-end, so that the resulted libraries can be faster and more lightweight. As an example, *TFLite* stems from *TensorFlow*, but is designed for edge devices and reported to have lower inference time and smaller library size than *TensorFlow* [106]. Besides those popular frameworks, we identify 34 (16.1%) DL apps that don't use any framework in Table 2. These apps use self-developed engines to support DL functionality.

• **Are mobile DL frameworks gaining traction?** As shown in Figure 4, DL frameworks optimized for smartphones, such as *TFLite* and *ncnn*, quickly gain popularity: the number of *TFLite*-based DL apps has increased from 12 to 31; that of *ncnn* increases from 21 to 28. We deem it as the trend of mobile DL ecosystem: *To train a model offline, use large, mature, and generic frameworks that focus on developer friendliness and feature completeness. To deploy a model on edge devices, switch to mobile-oriented frameworks that focus on performance (inference latency, memory footprint, and library size).*

Table 2: An overview of popular deep learning frameworks and their smartphone support at the time of writing (Nov. 2018).

Framework	Owner	Supported Mobile Platform	Mobile API	Is Open-source	Supported Model Format	Support Training
TensorFlow [36]	Google	Android CPU, iOS CPU	Java, C++	✓	ProtoBuf (*.pb, .pbtxt*)	✓
TF Lite [37]	Google	Android CPU, iOS CPU	Java, C++	✓	FlatBuffers (*.tflite*)	✗
Caffe [65]	Berkeley	Android CPU, iOS CPU	C++	✓	customized, json (*.caffemodel, .prototxt*)	✓
Caffe2 [9]	Facebook	Android CPU, iOS CPU	C++	✓	ProtoBuf (*.pb*)	✓
MxNet [46]	Apache Incubator	Android CPU, iOS CPU	C++	✓	customized, json (*.json, .params*)	✓
DeepLearning4J [13]	Skymind	Android CPU	Java	✓	customized (*.zip*)	✓
ncnn [35]	Tencent	Android CPU, iOS CPU	C++	✓	customized (*.params, .bin*)	✗
OpenCV [26]	OpenCV Team	Android CPU, iOS CPU	C++	✓	TesnorFlow, Caffe, etc	✗
FeatherCNN [16]	Tencent	Android CPU, iOS CPU	C++	✓	customized (*.feathermodel*)	✗
PaddlePaddle [24]	Baidu	Android CPU, iOS CPU & GPU	C++	✓	customized (*.tar*)	✓
xNN [40]	Alibaba	Android CPU, iOS CPU	unknown	✗	unknown	unknown
superid [34]	SuperID	Android CPU, iOS CPU	unknown	✗	unknown	unknown
Parrots [30]	SenseTime	Android CPU, iOS CPU	unknown	✗	unknown	unknown
MACE [23]	XiaoMi	Android CPU, GPU, DSP	C++	✓	customized (*.pb, .yml, .a*)	✗
SNPE [31]	Qualcomm	Qualcomm CPU, GPU, DSP	Java, C++	✗	customized (*.dlc*)	✗
CNNDroid [70]	Oskouei et al.	Android CPU & GPU	Java	✓	MessagePack (*.model*)	✗
CoreML [12]	Apple	iOS CPU, GPU	Swift, OC	✗	customized, ProtoBuf (*.proto, .mlmodel*)	✓
Chainer [10]	Preferred Networks	/	/	✓	customized (*.chainermodel*)	✓
CNTK [22]	Microsoft	/	/	✓	ProtoBuf (*.proto*)	✓
Torch [39]	Facebook	/	/	✓	customized (*.dat*)	✓
PyTorch [29]	Facebook	/	/	✓	customized, pickle (*.pkl*)	✓

We also investigate into the DL check-in and check-out behavior in mobile apps. We define the check-in DL apps as those that have no DL usage in earlier version (Jun. 2018) but add the DL usage in newer version (Sep. 2018), and the check-out vice versa. Note that the app list of our two datasets are not identical since we crawl the most popular ones, but the popularity is changing. So we only consider the apps that exist in both lists (11,710 in total), and conclude the results in Figure 5. As observed, 48 out of the 190 (25.3%) DL apps in newer version are checked in between Jun. 2018 and Sep. 2018. We also notice that some DL apps in old version checked out, but the number is much smaller (5). The reasons of check-out can be that the developers remove the corresponding functionality or just replace the DL with other approaches. Overall, the statistics support the fact that DL technique is increasingly adopted in mobile apps.

• **What is the storage overhead of frameworks?** Figure 6 shows the sizes of DL libs, i.e. the physical incarnation of DL frameworks. As shown, the average size of DL libs is 7.5MB, almost 6 times compared to the non-DL libs. Here, we only use the non-DL libs found within DL apps. The results show that DL libs are commonly heavier than non-DL libs, because implementing DL functionality, even without training backend, is quite complex. Looking into different frameworks, using *TensorFlow* and *Caffe* results in larger DL libs, i.e., 15.3MB and 10.1MB respectively, while others are all lower than 5MB. The reason is that mobile supports of *TensorFlow* and *Caffe* are ported from the original frameworks and substantially reuse the code base from them. However, these two frameworks are designed for distributed on-cloud DL. As comparison, other

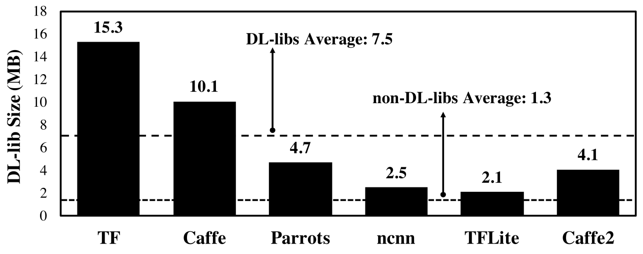

Figure 6: The binary library sizes of DL frameworks.

frameworks in Figure 6 are specifically designed for mobile devices to the purpose of good performance.

One app may incorporate multiple DL frameworks. Surprisingly, we find that 24 DL apps integrate more than one DL framework. For example, AliPay, the most popular payment app in China, has both *xnn* and *ncnn* inside. We deem such multi-usage as (potentially) bad practice, since it unnecessarily increases the apk size and memory footprint when these frameworks need to be loaded simultaneously. According to our statistics, the overhead is around 5.4MB, contributing to 13.6% to the total apk size on average. Such overhead can be avoided by running different tasks based on one framework, since most DL frameworks are quite generic and can support various types of DL models. Even if not, they can be easily extended to support the missing features [1]. The reason of such multi-usage behavior can be twofold. First, one app might be developed by different engineers (groups), who introduce different frameworks for their own DL purpose. Second, the developers may just reuse existing code and models for specific tasks, without merging them together in one DL implementation.

Implications: *Our findings highlight the advantages and popularity of mobile DL frameworks, encouraging further optimizations on them. Our findings also motivate app developers to give these frameworks priority considerations in choosing the incarnation of DL algorithms.*

5 MODEL ANALYSIS

This section focuses on the model-level analysis of DL technique. We first describe the methodology details, e.g., the design of Model Extractor in Section 5.1. Then, we show the analysis results on those DL models from three main aspects.

- The structures of DL models: the model types, layer types, and optimizations used (Section 5.2).
- The resource footprint of DL models: storage, memory, execution complexity, etc (Section 5.3).
- The security of DL models: using obfuscation and encryption to protect models from being stolen (Section 5.4).

5.1 Model Extractor: finding DL models

As a part of our analyzing tool (Section 3.2), Model Extractor takes DL apps which we have already identified as input, and outputs the DL model(s) used in each app. Model Extractor scans the *assets* folder of each decomposed DL apps, tries to validate each model file inside. Since DL frameworks use different formats to store their model files, Model Extractor has a validator for each of supported framework. However, we observe that many models are not stored as plaintext inside apk files. For example, some of them are encrypted on storage, and decrypted when apps running on devices. For such cases, Model Extractor tries to reverse engineer the apps, and extract the analyzable models.

Overall results We extract DL models based on the most popular frameworks, i.e., *TFLite*, *TensorFlow*, *ncnn*, *Caffe*, or *Caffe2*. In summary, we successfully extract 176 DL models, which come from 71 DL apps. The reasons why we cannot extract models from the other DL apps could be (i) the models are well protected and hidden in the apk files; (ii) the models are retrieved from Internet during runtime. Among the extracted models, we can analyze 98 of them, which come from 42 DL apps. The other extracted models cannot be parsed via our framework currently because (i) the models are in format which we have no insights into, such as *Parrots*-based models, since the corresponding frameworks are not open-source; (ii) the models are encrypted.

5.2 Model Structures

• **DL model types** Among the DL models extracted, 87.7% models are CNN models, 7.8% models are RNN models, while others are not confirmed yet. The CNN models are mostly used in image/video processing and text classification. The RNN models are mostly used in text/voice processing, such as word prediction, translation, speech recognition, etc. The results are consistent with the conventional wisdom: CNN models are good at capturing visual characteristics from images, while RNN models are powerful at processing sequential data with temporal relationships such as text and audio.

Table 3: Layers used in DL models. "% of models" shows how many models contain such layer, while "# in each model" shows the median/mean numbers of occurrences in each model that contains such layer. "conv" and "fc" are short for convolution and fully-connect.

Layer type	% of models	# in each model	Layer type	% of models	# in each model
conv	87.7	5 / 14.8	relu	51.0	6 / 16.3
pooling	76.5	2 / 2.8	split	46.9	1 / 7.5
softmax	69.1	1 / 1.1	prelu	32.1	4 / 4.6
fc	60.5	3 / 5.6	reshape	28.4	2 / 24.1
add	56.8	9.5 / 23.8	dropout	21.0	1 / 1.0

Table 4: Optimizations applied on DL models.

	1-bit Quan.	8-bit Quan.	16-bit Quan.	Sparsity
TF	unsupported	4.78%	0.00%	0.00%
TFLite	unsupported	66.67%	unsupported	unsupported
Caffe	unsupported	0.00%	unsupported	unsupported
Caffe2	unsupported	0.00%	unsupported	unsupported
ncnn	unsupported	0.00%	unsupported	unsupported
Total	*0.00%*	*6.32%*	*0.00%*	*0.00%*

• **DL layer types** We then characterize different types of layers used in DL models. As shown in Table 3, convolution (conv) is the most commonly used type. 87.7% models have at least one convolutional layer, and the median (mean) number of convolutional layers used in those models is 5 (14.8). This is not surprising since convolution is the core of CNN models, and CNN is the dominant model architecture used in vision tasks. Our previous analysis already demonstrates that image processing is the most popular use case of mobile DL. Similarly, pooling is also an important layer type in CNN models, included in 76.5% DL models. Besides, softmax is also frequently used (69.1%), but don't repeatedly show up in one single model. This is because softmax layer usually resides at the end of DL models to get the probabilities as output. As a comparison, fully-connected (fc) layers are less common, only used in 60.5% DL models. A possible reason is that fully-connected layer is known to be very parameter- and computation-intensive, and can be replaced by other layers such as convolution [17]. Other frequently used layer types include add, split, relu, prelu, dropout, and reshape.

Implications: *Our findings motivate framework and hardware vendors who are interested in optimizing mobile DL to focus on the popular DL layers that we discovered in deployed models, e.g. convolution.*

We also notice that a small number (5) of DL models contain customized layer types. Such customization is made as an extension to existing DL frameworks [1]. The result indicates that the functionalities of current DL frameworks are mostly complete enough.

• **Model optimizations** Various techniques have been proposed to optimize the DL models in consideration of their sizes and computation complexity. Here, we study the usage of two most popular techniques in the wild: quantization and sparsity. **Quantization** compresses DL models by reducing the number of bits required

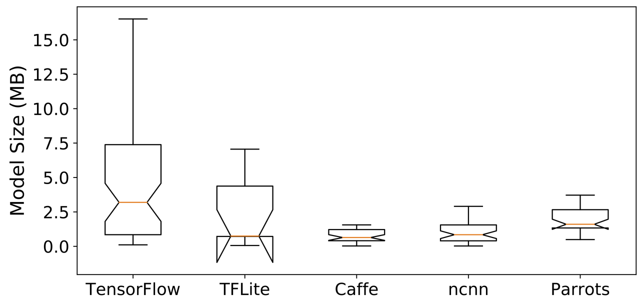

Figure 7: The size of DL models in different frameworks. We leave out *Caffe2* since we only successfully extract one model in *Caffe2* format.

to represent each weight. The quantization has different levels, including 16-bit [56], 8-bit [95], and even 1-bit [48, 84], while the original models are usually presented in 32-bit floating points. **Sparsity** [71, 96] has also been extensively studied in prior literatures as an effective approach to make compact DL models. It's mainly used in matrix multiplication and convolutional layers to reduce parameters. Such optimizations are known to reduce DL cost by up to two orders of magnitude without compromising model accuracy [58].

Table 4 summarizes the optimizations applied on DL models. Here we focus on the DL models for 5 popular frameworks, i.e., *TensorFlow (TF), TFLite, Caffe, Caffe2*, and *ncnn*. Overall, most of these frameworks only support 8-bit quantization, except *TensorFlow* who has 16-bit quantization and sparsity support. However, only a fraction of DL models apply the optimization techniques: 6.32% models are quantized into 8-bit, while others are non-optimized.

Implications: *The findings that well-known DL optimizations are missing in real-world deployment suggest the efficiency potential of mobile DL is still largely untapped. The findings also urge immediate actions to fix the missing optimizations.*

5.3 Model Resource Footprint

• **Model size.** Figure 7 illustrates the size of DL models (in storage). Overall, we find that the extracted DL models are quite small (median: 1.6MB, mean: 2.5MB), compared to classical models such as VGG-16 [90] (around 500MB) and MobileNet [63] (around 16MB). The models in *TensorFLow* format (median: 3.2MB) are relatively larger than the models in other formats such as *TFLite* (median: 0.75MB) and *ncnn* (median: 0.86MB).

• **Runtime overhead.** We then study the runtime performance of DL models. Here, we focus on two aspects: memory and computations. The memory usage includes both the model parameters and the generated intermediate results (feature maps). For computation complexity, we use floating point operations (FLOPs) during one inference as the metric. Here we use only part of models in *TensorFlow* and *ncnn* formats since some others don't have fixed input sizes, e.g., image size, so that the computation complexity can only be determined at runtime [15]. We also include the performance of some other classical CNN models such as AlexNet, MobileNet, etc.

As illustrated in Figure 8, the black crosses represent the DL models we have extracted, while the red dots represent the classical

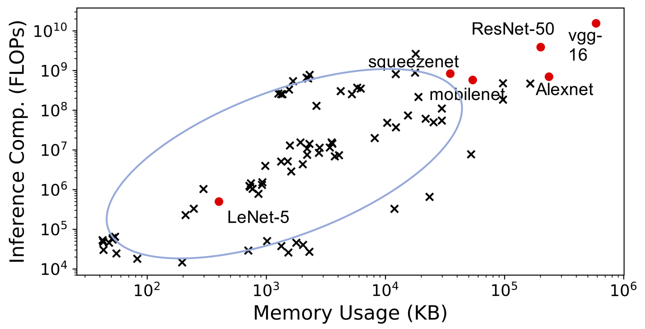

Figure 8: The cost of memory and computation of DL models extracted from apps. Red dots: classical CNN architectures as references. Black crosses: extracted DL models.

CNN architectures. Overall, the results show that *in-the-wild DL models are very lightweight in consideration of memory usage and computation complexity, with median value of 2.47 MB and 10M FLOPs respectively.* Running such models on mobile processors is inexpensive. For example, as estimated on the CPU of Snapdragon 845[3], the execution time of 80% models are less than 15ms which is translated to 67 FPS [32]. To be compared, ResNet-50, one of the state-of-the-art models in image classification task, has around 200MB memory usage and 4GFLOPs computations. Even MobileNet and SqueezeNet, which are designed and optimized for mobile scenarios, require more memory usage and computations than 90% those mobile DL models that we have discovered.

Implications: *Our findings of dominant lightweight DL models on smartphones give app developers a valuable assurance: DL inference can be as cheap as a few MBs of memory overhead and tens of ms execution delay. Our findings challenge existing research on DL inference, which are typically centered on full-blown models (e.g. VGG) and validated on these models. Given the significance of smartphones as DL platforms, future DL algorithm proposals should consider applicability on lightweight models and resource constraints concern.*

5.4 Model Security

Finally, we investigate into how DL models are protected. We deem model protection as an important step to AI system/app security, because if attackers can acquire the model, they can (i) steal the intellectual property by reusing the model file or re-train a new model; (ii) easily attack the DL functionality via adversarial attack [80]. We focus on two practical protection mechanisms.

- **Obfuscation** is a rather shallow approach to prevent attackers from gaining insights into the model structures by removing any meaningful text, e.g., layer names.
- **Encryption** is better in security by avoiding attackers from getting the model structures/parameters, but also causes inevitable overhead for apps to decrypt the models in memory. Here, we deem encrypted models as always obfuscated too.

We investigate into how obfuscation and encryption are employed on DL models that we have extracted. We analyze the DL models

[3]A typical mobile chip used by many popular smartphones such as Galaxy S8.

extracted from apps using *TensorFLow, TFLite, ncnn, caffe*, and *Caffe2*. In total, we confirm the security level of 120 DL models. Note that here encryption doesn't necessarily mean encryption algorithm, but also includes cases where developers customize the model format so that the model cannot be parsed via the DL framework.

The results show that among the 120 DL models, we find 47 (39.2%) models are obfuscated and 23 (19.2%) models are encrypted. Note that these two sets of apps are overlapped: encrypted apps are also obfuscated. The results indicate that *most DL models are exposed without protection, thus can be easily extracted and utilized by attackers.* In fact, only few frameworks in Table 2 support obfuscation, e.g., *ncnn* can convert models into binaries where text is all striped [33], and *Mace* can convert a model to C++ code [11]. What's worse, no framework provides help in model encryption as far as we know. Thus, developers have to implement their own encryption/decryption mechanism, which can impose non-trivial programming overhead.

Implications: *The grim situation of model security urges strong protection over proprietary models in a way similar to protecting copyright digital contents on smartphones [41, 78, 87, 104]. This necessitates a synergy among new tools, OS mechanisms and policies, and hardware mechanisms such as ARM TrustZone [7].*

6 DISCUSSIONS

Limitations of our analyzing tool Although we carefully design our analyzer to capture as many DL apps as possible, and involve a lot of manual efforts to validate the results, we can still have false identifications. For example, those DL apps that neither depend on any popular DL frameworks, nor have any string patterns in the native libraries, will be missed out. In addition, the apps that have integrated DL frameworks but don't really use them will be falsely classified as DL apps. For the first case, we plan to mine the code pattern of DL implementation and use the pattern to predict more DL apps that might involve DL. For the second one, we plan to further enhance our analyzer with advanced static analysis technique [42] so that it can detect whether the API calls (sinks) of DL libraries will be invoked or not. Our tool can be further enhanced via dynamic analysis, e.g., running the extracted models on smartphones and inspecting the system behavior.

More platforms In this work, we only analyze the adoption of DL on Android apps. Though Android is quite representative of the mobile ecosystem, more interesting findings might be made by expanding our study on other platforms such as iOS and Android Wear. We believe that comparing the DL adoption on different platforms can feed in more implications to researchers and developers.

7 RELATED WORK

Mobile DL Due to their ubiquitous nature, mobile devices can create countless opportunities for DL tasks. Researchers have built numerous novel applications based on DL [44, 68, 69, 76, 83, 98]. Besides, various optimization techniques have been proposed to reduce the overhead of DL on resource-constrained mobile devices, e.g., model compression [50, 59, 67, 74, 97], hardware customizations [45, 47, 57, 103], and cloud offloading [61, 66, 99, 105]. These studies are usually carried out under lab environments, based on classical models such as VGG and ResNet, in lack of real-world workloads and insights. Thus, our work is motivated by those enormous efforts that try to bring DL to mobile devices, and fill the gap between the academic literature and industry products.

ML/DL as cloud services Besides on-device fashion, the DL functionality, or in a broader scope of Machine Learning (ML), can also be accessed as cloud services. The service providers include Amazon [2], Google [19], Microsoft Azure [21], etc. There are some prior analyzing studies focusing on such MLaaS (machine learning as a service) platforms. Yao *et al.* [101] comprehensively investigate into effectiveness of popular MLaas systems, and find that with more user control comes greater risk. Some other literature studies [52, 88, 94] focus on the security issues of those platforms towards different types of attacks.

Empirical study of DL Prior empirical analysis mainly focuses on assisting developers to build better DL apps/systems/models. Zhang *et al.* [107] characterize the defects (bugs) in DL programs via mining the StackOverflow QA pages and GitHub projects. Consequently, the results are limited in only open-source, small-scale projects. Fawzi *et al.* [51] analyze the topology and geometry of the state-of-the-art deep networks, as well as their associated decision boundary. These studies mostly focus on classical and small-scale DL models proposed in previous literature, while our study mine the knowledge from large-scale in-the-wild mobile apps.

DL Model protection Some recent efforts have been investigated in protecting DL models. For example, various watermarking mechanisms [41, 78, 87, 104] have been proposed to protect intellectual property of DL models. This approach, however, cannot protect models from being extracted and attacked. As a closer step, some researchers [55, 60, 93] secure the DL systems/models based on secure execution environments (SEE) such as Intel SGX [20]. Some DL frameworks also provide mechanisms for model protection. For example, *Mace* [23] supports developers to convert models to C++ code [11]. However, our results show that a large number of DL models are exposed without secure protection.

8 CONCLUSIONS

In this work, we have carried out the first empirical study to understand how deep learning technique is adopted in real-world smartphones, as a bridge between the research and practice. By mining and analyzing large-scale Android apps based on a static tool, we have reached interesting and valuable findings. Our findings also provide strong and valuable implications to multiple stakeholders of the mobile ecosystem, including developers, hardware designers, and researchers.

ACKNOWLEDGMENT

The authors affiliated with Peking University were supported by the National Key R&D Program of China under the grant number 2018YFB1004800, the Beijing Municipal Science and Technology Project under the grant number Z171100005117002, the National Natural Science Foundation of China under grant numbers 61725201, 61528201, and 61529201. Felix Xiaozhu Lin was support in part by a Google faculty award.

REFERENCES

[1] 2018. Add a new op in TensorFlow. https://www.tensorflow.org/guide/extend/op.

[2] 2018. Amazon Machine Learning. https://aws.amazon.com/machine-learning.
[3] 2018. An Exploration of Mobile First AI. https://medium.com/swlh/an-exploration-of-mobile-first-ai-576c944efd36.
[4] 2018. Android aapt. http://elinux.org/Android_aapt.
[5] 2018. Apktool: A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.
[6] 2018. Apple COO: Smartphone is a 'major platform' for future of AI. https://www.techrepublic.com/article/apple-coo-smartphone-is-a-major-platform-for-future-of-ai/.
[7] 2018. Arm TrustZone. https://developer.arm.com/technologies/trustzone.
[8] 2018. Artificial Intelligence Next Key Growth Area for Smartphones as Numbers Top Six Billion by 2020, IHS Markit Says. https://news.ihsmarkit.com/press-release/technology/artificial-intelligence-next-key-growth-area-smartphones-numbers-top-six-bi.
[9] 2018. Caffe2 deep learning framework. https://github.com/caffe2/caffe2.
[10] 2018. Chainer. https://chainer.org/.
[11] 2018. Converting model to C++ code. https://mace.readthedocs.io/en/latest/user_guide/advanced_usage.html.
[12] 2018. CoreML by Apple. https://developer.apple.com/documentation/coreml.
[13] 2018. Deep Learning for Java. https://deeplearning4j.org/.
[14] 2018. dex2jar. https://github.com/pxb1988/dex2jar.
[15] 2018. Dynamic shapes in TensorFlow. https://www.tensorflow.org/guide/tensors.
[16] 2018. FeatherCNN. https://github.com/Tencent/FeatherCNN.
[17] 2018. Fully-connected Layers in Convolutional Neural Networks). https://cs231n.github.io/convolutional-networks/.
[18] 2018. Global mobile OS market share in sales to end users. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.
[19] 2018. Google Prediction API. https://cloud.google.com/prediction.
[20] 2018. Intel Software Guard Extensions. https://software.intel.com/en-us/sgx.
[21] 2018. Microsoft Azure ML Studio. https://azure.microsoft.com/en-us/services/machine-learning.
[22] 2018. Microsoft Cognitive Toolkit (CNTK). https://github.com/Microsoft/CNTK.
[23] 2018. Mobile AI Compute Engine. https://github.com/XiaoMi/mace.
[24] 2018. Mobile deep learning. https://github.com/baidu/mobile-deep-learning.
[25] 2018. Open neural network exchange format. https://onnx.ai/.
[26] 2018. Open Source Computer Vision Library. https://opencv.org/.
[27] 2018. Over Half of Smartphone Owners Use Voice Assistants. https://voicebot.ai/2018/04/03/over-half-of-smartphone-owners-use-voice-assistants-siri-leads-the-pack/.
[28] 2018. Protocol Buffer. https://developers.google.com/protocol-buffers/.
[29] 2018. pytorch. http://pytorch.org/.
[30] 2018. SenseTime. https://www.sensetime.com/?lang=en-us.
[31] 2018. Snapdragon Neural Processing Engine. https://developer.qualcomm.com/software/snapdragon-neural-processing-engine.
[32] 2018. Snapdragon performance. https://www.anandtech.com/show/12420/snapdragon-845-performance-preview/2.
[33] 2018. Strip visible string in ncnn. https://github.com/Tencent/ncnn/wiki/how-to-use-ncnn-with-alexnet.
[34] 2018. SuperID Android SDK. https://github.com/SuperID/superid-android-sdk.
[35] 2018. Tencent ncnn deep learning framework. https://github.com/Tencent/ncnn.
[36] 2018. TensorFlow. https://www.tensorflow.org/.
[37] 2018. TensorFlow Lite. https://www.tensorflow.org/mobile/tflite/.
[38] 2018. The Machine Learning Behind Android Smart Linkify. https://ai.googleblog.com/2018/08/the-machine-learning-behind-android.html.
[39] 2018. torch. http://torch.ch/.
[40] 2018. xNN deep learning framework. https://myrgzn.gitee.io/rgzn/news/page100.html.
[41] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. 2018. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. *arXiv preprint arXiv:1802.04633* (2018).
[42] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[43] Davide Bacciu, Stefano Chessa, Claudio Gallicchio, and Alessio Micheli. 2017. On the need of machine learning as a service for the internet of things. In *Proceedings of the 1st International Conference on Internet of Things and Machine Learning, IML 2017*. 22:1–22:8.
[44] Michael Barz and Daniel Sonntag. 2016. Gaze-guided Object Classification Using Deep Neural Networks for Attention-based Computing. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'16)*. 253–256.
[45] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a Small-footprint High-throughput Accelerator for Ubiquitous Machine-Learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 269–284.
[46] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015).
[47] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*. 367–379.
[48] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.
[49] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. 191–198.
[50] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems (NIPS'14)*. 1269–1277.
[51] Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, Pascal Frossard, and Stefano Soatto. 2018. Empirical study of the topology and geometry of deep networks. In *IEEE CVPR*.
[52] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1322–1333.
[53] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.
[54] Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 311–320.
[55] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. 2018. Securing Input Data of Deep Learning Inference Systems via Partitioned Enclave Execution. *arXiv preprint arXiv:1807.00969* (2018).
[56] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.
[57] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA'16)*. 243–254.
[58] 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).
[59] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'16)*. 123–136.
[60] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. 2018. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *arXiv preprint arXiv:1808.00590* (2018).
[61] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor N. Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 27–40.
[62] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. 2018. Privacy-preserving Machine Learning as a Service. *PoPETs* 2018, 3 (2018), 123–142.
[63] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017).
[64] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <1MB Model Size. *arXiv preprint arXiv:1602.07360* (2016).
[65] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*. 675–678.
[66] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor N. 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 (ASPLOS'17)*. 615–629.

[67] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-power Deep Learning Inference on Mobile Devices. In *15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2016)*. 23:1–23:12.

[68] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. 2015. 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 (IoT-App'15)*. 7–12.

[69] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. 2015. 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)*. 283–294.

[70] Seyyed Salar Latifi Oskouei, Hossein Golestani, Matin Hashemi, and Soheil Ghiasi. 2016. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. In *Proceedings of the 2016 ACM on Multimedia Conference*. 1201–1205.

[71] Vadim Lebedev and Victor Lempitsky. 2016. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2554–2564.

[72] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[73] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th international conference on Knowledge discovery and data mining (KDD'14)*. 661–670.

[74] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18)*. 389–400.

[75] Xuanzhe Liu, Huoran Li, Xuan Lu, Tao Xie, Qiaozhu Mei, Feng Feng, and Hong Mei. 2018. Understanding Diverse Usage Patterns from Large-Scale Appstore-Service Profiles. *IEEE Trans. Software Eng.* 44, 4 (2018), 384–411.

[76] Gaurav Mittal, Kaushal B. Yagnik, Mohit Garg, and Narayanan C. Krishnan. 2016. 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)*. 940–945.

[77] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[78] Yuki Nagai, Yusuke Uchida, Shigeyuki Sakazawa, and ShinâĂŽichi Satoh. 2018. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval* 7, 1 (2018), 3–16.

[79] Shumpei Okura, Yukihiro Tagami, Shingo Ono, and Akira Tajima. 2017. Embedding-based news recommendation for millions of users. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1933–1942.

[80] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. 372–387.

[81] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'14)*. 701–710.

[82] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[83] Valentin Radu, Nicholas D. Lane, Sourav Bhattacharya, Cecilia Mascolo, Mahesh K. Marina, and Fahim Kawsar. 2016. Towards Multimodal Deep Learning for Activity Recognition on Mobile Devices. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'16)*. 185–188.

[84] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. 525–542.

[85] Mauro Ribeiro, Katarina Grolinger, and Miriam A. M. Capretz. 2015. MLaaS: Machine Learning as a Service. In *14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015*. 896–902.

[86] Everett M Rogers. 2010. *Diffusion of innovations*. Simon and Schuster.

[87] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. [n. d.]. DeepSigns: A Generic Watermarking Framework for Protecting the Ownership of Deep Learning Models. ([n. d.]).

[88] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on*. 3–18.

[89] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354.

[90] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[91] SÃķren Sonnenburg, Mikio L Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoffrey Holmes, Yann LeCun, Klaus-Robert MÃžller, Fernando Pereira, Carl Edward Rasmussen, et al. 2007. The need for open source software in machine learning. *Journal of Machine Learning Research (JMLR)* 8, Oct (2007), 2443–2466.

[92] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. 2017. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855* (2017).

[93] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. *arXiv preprint arXiv:1806.03287* (2018).

[94] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs.. In *USENIX Security Symposium*. 601–618.

[95] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. 4.

[96] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. 2074–2082.

[97] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, (CVPR'16)*. 4820–4828.

[98] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. 2018. DeepType: On-Device Deep Learning for Input Personalization Service with Minimal Privacy Concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (2018), 197.

[99] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. 2019. DeepWear: Adaptive Local Offloading for On-Wearable Deep Learning. *IEEE Transactions on Mobile Computing* (2019).

[100] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 129–144.

[101] Yuanshun Yao, Zhujun Xiao, Bolun Wang, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. 2017. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*. 384–397.

[102] Xiao Zeng, Kai Cao, and Mi Zhang. 2017. *MobileDeepPill*: A Small-Footprint Mobile Deep Learning System for Recognizing Unconstrained Pill Images. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*. 56–67.

[103] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. 161–170.

[104] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. 2018. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 159–172.

[105] Qingchen Zhang, Laurence T. Yang, and Zhikui Chen. 2016. Privacy Preserving Deep Computation Model on Cloud for Big Data Feature Learning. *IEEE Trans. Computers* (2016), 1351–1362.

[106] Xingzhou Zhang, Yifan Wang, and Weisong Shi. 2018. pCAMP: Performance Comparison of Machine Learning Packages on the Edges. (2018).

[107] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. (2018).