An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications

Zhenpeng Chen*, Huihan Yao*, Yiling Lou*, Yanbin Cao*†, Yuanqiang Liu*, Haoyu Wang‡, and Xuanzhe Liu*

*Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China
†Peking University Information Technology Institute (Tianjin Binhai), Tianjin, China
‡Beijing University of Posts and Telecommunications, Beijing, China
{czp, yaohuihan, yiling.lou, caoyanbin, yuanqiangliu}@pku.edu.cn, haoyuwang@bupt.edu.cn, xzl@pku.edu.cn

Abstract—Deep learning (DL) is moving its step into a growing number of mobile software applications. These software applications, named as DL based mobile applications (abbreviated as *mobile DL apps*) integrate DL models trained using large-scale data with DL programs. A DL program encodes the structure of a desirable DL model and the process by which the model is trained using training data. Due to the increasing dependency of current mobile apps on DL, software engineering (SE) for mobile DL apps has become important. However, existing efforts in SE research community mainly focus on the development of DL models and extensively analyze faults in DL programs. In contrast, faults related to the deployment of DL models on mobile devices (named as *deployment faults of mobile DL apps*) have not been well studied. Since mobile DL apps have been used by billions of end users daily for various purposes including for safety-critical scenarios, characterizing their deployment faults is of enormous importance. To fill in the knowledge gap, this paper presents the first comprehensive study to date on the deployment faults of mobile DL apps. We identify 304 real deployment faults from Stack Overflow and GitHub, two commonly used data sources for studying software faults. Based on the identified faults, we construct a fine-granularity taxonomy consisting of 23 categories regarding to fault symptoms and distill common fix strategies for different fault symptoms. Furthermore, we suggest actionable implications and research avenues that can potentially facilitate the deployment of DL models on mobile devices.

Index Terms—deep learning, mobile applications, deployment faults

I. INTRODUCTION

In recent years, deep learning (DL) has emerged as one of the most popular and promising techniques and has been widely adopted in various applications [1]–[5]. Mobile devices are undoubtedly among the most important platforms for running DL based software applications [6]–[8]. These software applications, namely DL based mobile applications (in short as *mobile DL apps*), integrate DL capabilities to add a wide range of features including object detection [9], image processing [10], natural language processing [11], speech recognition [12], etc. To achieve this goal, developers train DL models using large-scale data (i.e., *development of DL models*), and then deploy the obtained DL models on mobile devices for real usage (i.e., *deployment of DL models*).

In fact, *development of DL models* is a general process for different types of DL based applications [13] and its challenges

Corresponding author: Xuanzhe Liu (xzl@pku.edu.cn).

have been well studied in the software engineering (SE) research community [14]–[19]. In particular, researchers [16]–[19] have extensively analyzed faults in the DL programs written based on DL frameworks (e.g., TensorFlow (TF) [20] and Keras [21]), which encode the structure of desirable DL models and the process by which the models are trained using the training data.

Recently, the rapid growth of mobile DL apps [22] has posed urgent challenges to the *deployment of DL models*, i.e., deploying DL models on mobile devices. For example, computation-intensive DL models can be executed efficiently on PC/server platforms, but they cannot be directly deployed and executed on mobile devices with limited computing power [23]. Although major vendors have rolled out specific DL frameworks such as TF Lite [24] and Core ML [25] to facilitate this deployment process, various specific faults are still emerging in this process and frequently asked on Stack Overflow (SO), one of the most popular Q&A forums for developers [13]. Moreover, previous work [13] has demonstrated that relevant questions are increasing rapidly on SO and more difficult to resolve than those related to other aspects of DL based applications. In addition, mobile DL apps are not only used by billions of end users for their daily activities (e.g., speech-to-text and photo beauty) [22], [26], but also reported to be increasingly adopted in various safety-critical scenarios (e.g., driver assistance [27] and autonomous vehicles [28]). Therefore, the emerging faults related to the deployment of DL models on mobile devices (named as *deployment faults of mobile DL apps*) should be carefully addressed. Unfortunately, the characteristics of these faults have not been well understood.

To fill in the knowledge gap, this paper presents the first comprehensive study on analyzing *symptoms* and *fix strategies* of deployment faults of mobile DL apps. Given the surging popularity of mobile DL apps, this study is of enormous importance. It can help in understanding what are the common deployment faults of mobile DL apps and how these faults are resolved in practice, so as to provide a high-level categorization that can serve as a guide for developers to resolve common faults and for researchers to develop tools for detecting and fixing deployment faults of the increasing mobile DL apps.

We focus our study on the faults that occur during the usage of two representative frameworks specifically designed for deploying DL models on mobile devices, i.e., TF Lite [24] and Core ML [25], both of which are widely used in industry practice and well adopted in related studies [13], [23]. Specifically, we collect a dataset of 304 deployment faults related to their usage from SO and GitHub, two commonly used data sources for studying software faults [16]–[19], [29].

By manual analysis, we qualitatively extract the symptom of each identified fault and construct a hierarchical taxonomy containing 23 symptom categories, indicating the diversity of deployment faults of mobile DL apps. Additionally, we distill common fix strategies for each symptom category, providing insights about deployment fault resolution of mobile DL apps. Based on our results, we discuss new directions for future research. Furthermore, we offer the scripts and the data used in this study [30] as an additional contribution to the research community for other researchers to replicate and build upon.

II. Background and Research Questions

We start by introducing current practice of the development of DL models and the deployment of DL models on mobile devices. Fig. 1 distinguishes the two processes.

Development of DL models. Development of DL models is a general process for different types of DL based software applications [13]. First, developers construct structures of desirable DL models and specify run-time configuration (e.g., hyper-parameters) with DL programs written based on state-of-the-art DL frameworks such as TF and Keras. A DL model consists of multiple layers to convert input to output, with each layer containing a set of neurons that accept input from neurons in the preceding layer, apply activation function to the input, and pass the resulting output to the neurons in the succeeding layer via a set of weighted edges. The layer used as an entry point into the DL model is called the input layer, while the layer that produces the end result is called the output layer. The input and output layers wrap the input and output tensors (multi-dimensional arrays of numerical values), respectively. Then, developers use large-scale data to train the DL models, during which the weights of edges in the models are adjusted and set to values that minimize the difference between model output and expected output. Finally, developers evaluate the performance (e.g., accuracy) of the obtained DL models using testing data. Due to space limit, we present only the model training phase in Fig. 1.

Deployment of DL models. DL models, which are demonstrated to meet the performance requirements, are ready to be deployed on mobile devices for real usage. The deployment process mainly focuses on platform adaptations. Due to the limited computing power, memory size, and energy capacity of mobile devices, models trained on PC/server platforms cannot be directly deployed on them. To tackle this problem, some lightweight frameworks, such as TF Lite for Android and Core ML for iOS, are specifically designed for converting trained DL models to the formats supported by mobile devices. Specifically, Core ML provides Python APIs for this task,
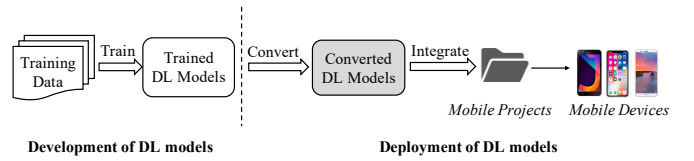

Fig. 1. Development and deployment of DL models

while TF Lite provides both CLIs and Python APIs. It is a common practice in the conversion stage to perform model quantization to reduce precision representations of the weights of edges in trained DL models, in order to reduce memory cost and computing overhead. For example, Core ML supports converting the weights from 32 bits to 16/8/4 bits. Then, developers can integrate the converted models into mobile projects with the help of TF Lite and Core ML. For instance, TF Lite provides APIs of various programming languages, such as Java, C++, and Python, to support the integration. Finally, the integrated projects can run on mobile devices and make inference based on input data.

Scope and research questions. We focus our analysis on the process of deploying DL models to mobile devices. Any faults related to this process are within our scope. However, faults that occur during the development of DL models are not considered in this study. Specifically, we aim to address two research questions that are concerned with deployment faults of mobile DL apps:

RQ1 (Symptoms): *What are the frequent fault symptoms?*

RQ2 (Fix strategies): *What are the common fix strategies for different fault symptoms?*

III. Methodology

To characterize the deployment faults of mobile DL apps, we analyze the relevant questions posted on SO and the relevant issues posted on GitHub. We illustrate the overview of the methodology of our study in Fig. 2.
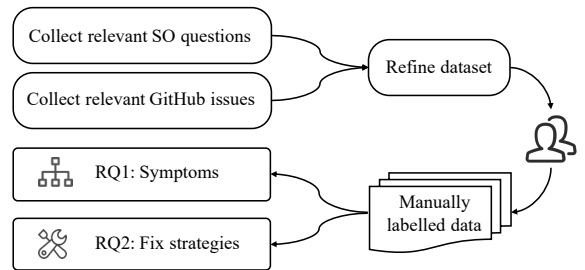

Fig. 2. Overview of the methodology.

A. Data Collection

Following previous studies [13], [23], we focus on two representative DL frameworks (i.e., TF Lite and Core ML) that are specially designed for deploying DL models on mobile devices. Since the deployment process is supported by these frameworks, we collect the faults that occur in their usage to construct the dataset of interest.

1) Mining SO: As one of the most popular community-driven Q&A websites, SO's users range from novices to experts [16], increasing the diversity of our collected faults. In addition, developers often post questions on SO for the faults that they cannot find solutions quickly, leading to more non-trivial faults in our dataset. We collect the relevant questions on SO in the following steps.

Download SO dataset. We first download the entire SO dataset from the official Stack Exchange Data Dump [31] on June 7, 2020. The dataset covers the SO posts generated from July 31, 2008 to June 2, 2020. Each SO question has one to five tags based on its topics.

Extract candidate posts. We then extract SO questions tagged with TF Lite and Core ML. In line with previous work [16], [17], we filter out the questions that do not contain any source code because questions about faults usually contain code snippets. In addition, we follow previous studies [18], [32] to exclude questions that do not have an accepted answer, ensuring that we consider only questions with a confirmed solution. As a result, we obtain 154 questions for TF Lite and 149 questions for Core ML.

2) Mining GitHub: In addition to SO, GitHub is also a commonly used data source for studying faults. Following previous work [29], we mine issues in the official GitHub repositories of the selected frameworks to identify faults that occur during their usage. Compared to commits, issues contain more fault information that includes original reports and developers' discussions [29]. Such a benign characteristic makes issues suitable for studying fault symptoms and fix strategies. In practice, we use the GitHub search API [33] to mine the issues about TF Lite and Core ML on June 27, 2020. Note that, on GitHub, issues are used for various purposes, including bug report, feature request, etc. To categorize the purposes of issues, developers often employ repository-specific keywords to label issues. In line with previous work [29], we also employ the issue labels to help us filter out irrelevant issues. The collection processes for TF Lite and Core ML are conducted separately as follows.

Extract issues for TF Lite. Since TF Lite has been integrated into the TF ecosystem, to obtain issues for TF Lite, we limit the search to issues in the official TF repository [34]. The first two authors jointly examine each label in the TF repository to determine which labels can be used for filtering. Then, we collect TF Lite related issues by extracting issues labeled with "*comp:lite*." Moreover, we filter out issues labeled with "*type:feature*," "*type:bug*," "*type:docs-bug*," "*type:docs-feature*," or "*type:build/install*" to exclude those about requests for new features, bugs in the framework itself, document-related problems, and requests for framework installment/build instructions. To ensure that we consider only issues with a confirmed solution, we further exclude those without answers or responses (i.e., those labeled with "*stalled*" or "*stat:awaiting response*"). Overall, we obtain 626 issues for TF Lite.

Extract issues for Core ML. To obtain issues for Core ML, we first extract all the issues in the official Core ML repository [35]. Since labels in this repository are not as abundant as those in the TF repository, with the help of issue labels, we can filter out only the issues about bugs in the framework itself (i.e., those labeled with "*label:bug*"). Then, similar to the process for TF Lite, we extract only the closed issues. Overall, we obtain 169 issues for Core ML.

3) Refining Dataset: Since the extracted posts (i.e., questions and issues) may contain some noise that is not about faults (e.g., how-to questions on SO), the third and fourth authors further filter the extracted posts through manual analysis. Specifically, they jointly read the extracted posts and exclude any post that either is not related to any issue-fixing activity or happens to fix an issue in the framework itself rather than in mobile DL apps. During this process, any conflicts are discussed and resolved by introducing an arbitrator, who has three years of experience in deploying DL models on mobile devices and has published several papers related to this topic in top-tier conferences. Finally, for TF Lite, we have 65 SO questions and 132 GitHub issues; for Core ML, we have 52 SO questions and 38 GitHub issues.

B. Manual Labelling

The refined dataset, which consists of 287 posts, is used for distilling symptoms and fix strategies through manual labelling. The scale of this dataset is comparable and even larger than those used in existing fault-related studies [15], [16], [29], [36], [37] that also require manual inspection. Next, we present our procedures of manual labelling.

1) Pilot Labelling: First, we randomly sample 50% of the 287 posts for a pilot labelling. The first two authors, who have five and three years of DL experience respectively, jointly participate in the process. They follow an open coding procedure [38] to inductively create categories for symptoms and fix strategies by analyzing the sampled posts. The detailed procedures are described below.

The two authors read and reread all the posts to understand the context of faults and assign each post with short but descriptive phrases as initial codes to indicate (i) the *fault symptom* that shows what the fault looks like and (ii) the *fix strategy* that tells how a fault is fixed. In this process, they take all the contents of each post, including the title, description, code snippets, error messages, comments, answers, and even URLs mentioned by developers, for careful inspection.

Then, they proceed to construct taxonomies for symptoms and fix strategies, respectively. Specifically, they group similar codes into categories and the grouping process is iterative, in which they continuously go back and forth between categories and posts to refine the taxonomies. A post is assigned to all related categories if it is related to multiple faults. In the cases where there is no agreement between the two authors, the aforementioned arbitrator is introduced to make discussions and resolve the conflicts. They follow the procedure until they reach agreement on all posts.

2) Reliability Analysis: For reliability analysis, the first two authors then independently label the remaining 50% posts based on the coding schema generated in the pilot labelling. Specifically, they label each post with identified symptom

and fix strategy categories and add the posts that cannot be classified into the current taxonomies into a new category named *Pending*. To measure the inter-rater agreement during the independent labelling, we employ the widely used Cohen's Kappa (κ) [39] as the indicator. The κ values obtained for symptoms and fix strategies are 0.819 and 0.743, indicating almost perfect agreement and substantial agreement [40], respectively. The agreement levels demonstrate the reliability of our coding schema and procedure.

The conflicts of labelling are then discussed and resolved by the aforementioned arbitrator. For the posts classified as *Pending*, we also employ the arbitrator to help us further identify symptoms and fix strategies behind them and determine if new categories need to be added. As a result, we add three new categories into the symptom taxonomy and two new categories into the fix strategy taxonomy, and assign all the posts in *Pending* into the taxonomies. The final labelling results are checked and approved by all participants.

In summary, among the 287 posts, we identify a total of 304 faults. The labelling results in pilot labelling and reliability analysis are both included in the final taxonomies. Based on the taxonomies for symptoms and fix strategies, we answer the **RQ1** and **RQ2** raised in Section II, respectively.

IV. RQ1: SYMPTOMS

Fig. 3 presents the hierarchical taxonomy of deployment fault symptoms of mobile DL apps. The taxonomy is organized into three-level categories, including a root category (i.e., *Deployment Faults*), five inner categories linked to stages in deploying DL models (e.g., *Model Conversion*), and 23 specific leaf categories (e.g., *Model parse failure*).

Finding 1: *We construct a taxonomy of 23 fault symptom categories related to deploying DL models on mobile devices, indicating the diversity of deployment faults.*

For each category, the number in the top right corner refers to the number of faults in it. Due to space limit, we address only frequent and non-trivial symptoms (i.e., #faults $\geq$ 3). For *Data Preparation* and *Model Update*, we do not present their leaf categories since no frequent symptoms are observed under them. For the remaining three inner categories, faults with infrequent or unclear symptoms are included in the *Others* category. Next, we discuss and exemplify each inner category.

A. Model Conversion

As the first stage of deploying DL models, model conversion aims to convert DL models into the formats expected by mobile devices. To implement a converter for model conversion, developers need to provide the DL model that is ready to be converted and specify necessary information about the model through APIs/CLIs provided by TF Lite or Core ML. We observe 147 faults that occur during the model conversion stage, accounting for 48.4% of all the identified faults and covering 12 symptom categories.

A large proportion of faults occur when the converter parses the DL model and validates the model information specified by developers, such as names and shapes of input/output

tensors of the model. Specifically, 9.5% of faults in *Model Conversion* are triggered when the converter fails in parsing the DL model (*A.1*). Moreover, when the converter detects missing or incorrect specification of the aforementioned model information, developers may encounter *Tensor error (A.2)* and *Shape/size error (A.3)*. Furthermore, *Shape/size error (A.3)* can also be triggered when the converter detects the invalid shape of input/output tensors or the dimension/size misalignment in the model structure. In total, *A.2* and *A.3* account for 23.1% of faults in *Model Conversion*. In addition to the basic model information, developers can also specify some information to reduce the precision representations of model weights during the conversion stage, so as to reduce the memory cost and computing overhead of DL models on mobile devices. This process is commonly known as model quantization [13]. The problematic configuration of quantization-related arguments may result in two types of symptoms, i.e., *Quantization failure (A.4)* and *Unexpected model size (A.5)*, accounting for 6 out of the 147 faults (4.1%) in *Model Conversion*.

After parsing the DL model, the converter may find that the model uses operations or datatypes that are not supported by TF Lite or Core ML. This can result in *Unsupported operation (A.6)* and *Unsupported datatype (A.7)*, accounting for 31.3% and 3.4% of faults in *Model Conversion*, respectively. In particular, *A.6* is the most frequent category in the model conversion stage. Its common occurrence is because that compared to the frameworks used for developing DL models (e.g., TF and Keras), TF Lite and Core ML are proposed later and relatively unfledged. Therefore, some standard operators, functions, or layers (collectively referred to as "operations" here) used in the model may be not supported by TF Lite and Core ML. Moreover, the DL model may contain some custom operations that cannot be recognized by the converter.

In addition to the symptoms specific to the deployment of DL models, we also observe that a portion (i.e., 15.6%) of faults in *Model Conversion* share common symptoms with general software systems. For example, 4.8% of faults are triggered due to unsuccessful import of dependent modules (i.e., *Import error (A.8)*); 8.8% are related to reference to non-existent variables or functions (i.e., *Attribute not found (A.9)*); and 2.0% are caused by using arguments of API/CLI incorrectly (i.e., *Invalid argument (A.10)*).

Besides the faults with explicit errors thrown during the model conversion stage, sometimes developers get unexpected models even after model conversion appears to be successfully done. For example, developers may find that the number, shape, or format of input/output tensors of the model changes. We classify these cases into the category *Unexpected model (A.11)*, accounting for 4.1% faults in *Model Conversion*.

Finding 2: *Most (i.e., 48.4%) of deployment faults occur during the model conversion stage, covering a wide spectrum of symptoms (i.e., 12 categories). Among these categories, unsupported operation is the most common, accounting for 31.3% of faults in this stage.*
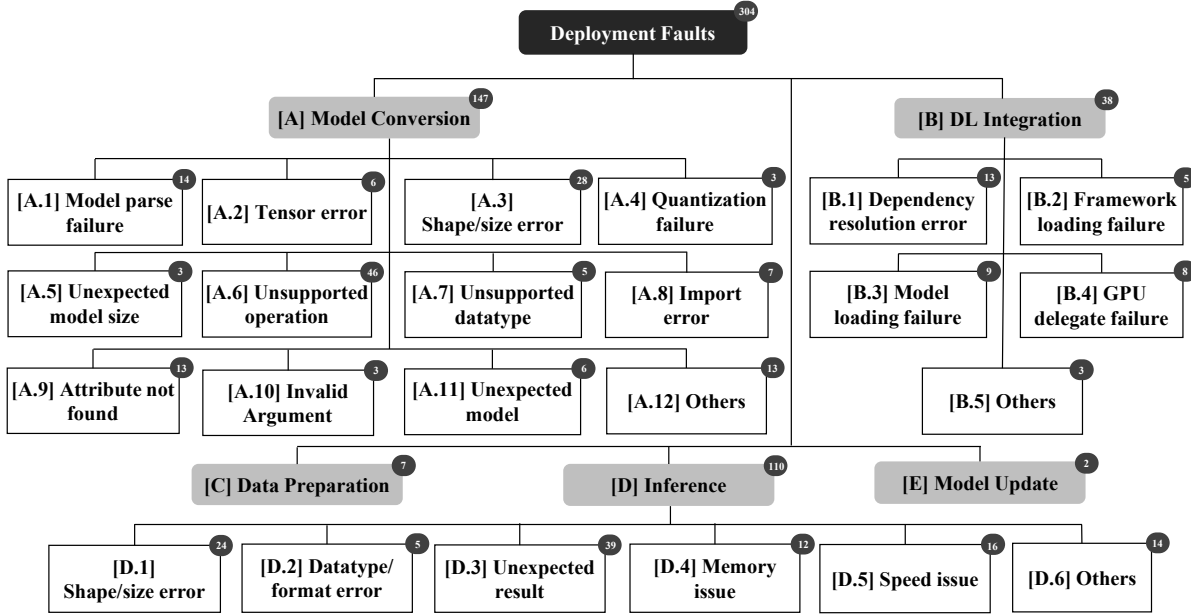
Fig. 3. Taxonomy of deployment fault symptoms of mobile DL apps.

B. DL Integration

After the DL model is converted into the expected format, developers can integrate it as well as DL frameworks into a mobile app project. Then, they can build the project and load the model to make it ready for inference. Faults that appear in this stage are included in the *DL Integration (B)* category, accounting for 12.5% of the deployment faults of mobile DL apps.

Dependency resolution error (B.1) is a common fault when building projects, accounting for 34.2% of the faults in *DL Integration*. Specifically, it refers to failures in preparing necessary dependencies directly or transitively specified by developers. In these cases, projects throw error messages like inability to resolve libraries, unsuccessful dependency downloading, and undefined reference to objects (e.g., functions and libraries).

After building projects, developers can run mobile apps to make it predictable. However, in this phase, many developers encounter *Framework loading failure (B.2)* and *Model loading failure (B.3)*, which refer to the failures in loading DL frameworks and models respectively and account for a total of 36.8% of faults in *DL Integration*. What is more, developers may configure projects to make it able to use the GPU backend on mobile devices. However, some developers complain that they encounter the *GPU delegate failure (B.4)* when running mobile DL apps. *B.4* represents 21.1% of faults in *DL Integration*.

Finding 3: *Faults appearing in the DL integration stage account for 12.5% of the total deployment faults and cover five symptom categories. A large proportion (34.2%) of these faults are thrown with dependency resolution errors.*

C. Data Preparation

Data Preparation (C) is the stage where a mobile app prepares input data for the next inference stage. For a mobile DL app, input data are usually extracted from user-generated data such as camera pictures or typed texts, and a data preparation fault often occurs when the app fails to access or process the required user-generated data. Note that this type of faults is essentially related to data accessing and processing issues, which not only occur in mobile DL apps, but also is very common in other mobile apps. Therefore, to seek more extensive help, developers usually do not describe these problems in the context of mobile DL apps (e.g., on SO they prefer not to post their problems with any tag related to DL), and thus we observe only a few related cases (2.3%) with no frequent symptoms in the collected data.

D. Inference

Inference (D) consists of faults that occur when a mobile app makes inference based on input data. 36.2% of deployment faults do not show symptoms until this stage.

A proportion (26.4%) of faults in *Inference* appear with explicit errors, i.e., *Shape/size error (D.1)* or *Datatype/format error (D.2)*. They are triggered when the shape/size or datatype/format of input/output arrays used for storing input/output data does not align with that of input/output tensors of the DL model.

Furthermore, some developers report that the mobile DL app produces unexpected results (i.e., *D.3*) although no errors are thrown. These cases account for 35.5% in *Inference*. Specifically, developers may observe that the mobile DL app produces different results than the original model. However, note that this symptom cannot be always used as the indication of faults, especially when model quantization is performed

during the model conversion stage. Since model quantization reduces the precision representations of model weights, it is reasonable to observe the change in model performance. Besides, developers also employ some other indications to confirm the existence of *Unexpected result (D.3)*. For instance, the mobile DL app produces the same result for any input or produces different results for the same input.

In addition to the faults that affect the output results, there are also 25.5% of faults that have impact on the memory usage and inference speed of mobile DL apps. We use *Memory issue (D.4)* and *Speed issue (D.5)* to refer to the two types of faults. Specifically, *Memory issue (D.4)* includes symptoms such as out of memory, memory leak, failures in memory allocation, and segment faults; *Speed issue (D.5)* is mainly manifested as long latency time of making inference.

Finding 4: *36.2% of faults occur when mobile DL apps make inference based on input data, covering six symptom categories. In particular, 35.5% of the faults in this stage are captured since developers observe unexpected results.*

E. Model Update

Once put into real usage, mobile DL apps keep receiving feedback from users (e.g., bad cases), based on which DL models can further be improved (e.g., updating the weights of models). Instead of re-training DL models on PC/server platforms and then re-deploying the new models again, developers can also directly re-train the DL models on mobile devices, which is the stage *Model Update (E)*. However, since currently on-device training requires a large amount of computational resources and is still not widely supported by existing DL frameworks, we observe only a few instances (0.7%) related to it in our dataset.

F. Distribution of Symptoms across Frameworks

We then further analyze the distribution of the identified fault symptoms across the two selected frameworks (i.e., TF Lite and Core ML). We find that the two frameworks share a similar distribution in most categories. For example, 37% of TF Lite related issues are included in *Inference (D)*, and for Core-ML, the ratio (35%) is comparable. However, at the same time, the two frameworks differ obviously in some specific categories. For instance, *Unsupported operation (A.6)* accounts for 18% of TF Lite related issues, but only 10% of Core ML related issues.

V. RQ2: FIX STRATEGIES

To capture how developers fix different types of deployment faults, for each symptom category, we summarize its fix strategies in this section. Since *Data Preparation* and *Model Update* contain only a few samples and do not show frequent symptoms, here, we do not consider them. For the remaining three inner categories, we show the frequency of different fix strategies on their leaf categories in Figs. 4, 5, and 6, respectively. Due to space limit, strategies with low frequency (i.e., #faults < 3) are not shown in the figures. In each figure, X axis represents each leaf category and the letter identifier

is consistent with our taxonomy in Fig. 3; Y axis shows fix strategies following with their total frequency under the inner category. Next, we elaborate the identified fix strategies for frequent symptoms and demonstrate some real-world examples of faults and corresponding fixes.

A. Fix Strategies for Faults in Model Conversion

We identify nine frequent fix strategies for faults in *Model Conversion* and illustrate the distribution of these strategies on leaf categories in Fig. 4.
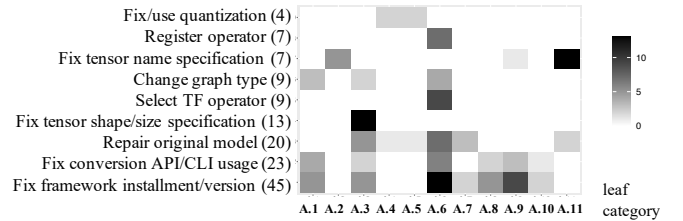

Fig. 4. Distribution of fix strategies for leaf categories in *Model Conversion*.

Fix framework installment/version. 30.6% of faults in *Model Conversion* are solved by re-installing the DL framework or switching the DL framework into a different version. This strategy covers seven fault symptoms, and is especially frequently adopted in the *Unsupported operation (A.6)* and *Attribute not found (A.9)* categories. For example, 36.1% of *Unsupported operation (A.6)* faults are fixed after switching the DL framework into a more recent version with more supported operations. As for *Attribute not found (A.9)* faults, developers often misuse APIs in a way unsupported by the current DL framework, since APIs frequently evolve with DL frameworks. Therefore, at most cases, developers resolve them by changing the DL framework to another version that supports the reference to specified attributes. For example, a developer reports that she receives an error "*AttributeError: type object TFLiteConverter has no attribute from_keras_model*" when converting a Keras model to the TF Lite format (TF issue #38786), and the corresponding fix is upgrading TF to 2.x version since *from_keras_model* is not supported by 1.x version. In addition, the framework version issue can also result in some non-intuitive symptoms. For example, a developer encounters a *Shape/size error (A.3)* during model conversion with the message "*Check failed: input_shape.dims().size() == op→size.size() (4 vs. 3)*" (SO post #56631820), which leads to a heated discussion. All the comments suggest that the developer should fix the shape of the input tensor specified during model conversion, but none of them work. Finally, the developer upgrades TF and successfully resolves the fault.

Fix conversion API/CLI usage. 15.6% of faults in *Model Conversion*, involving six frequent symptom categories, are fixed by correcting or changing the usage of APIs/CLIs for model conversion. As suggested by previous work [13], so many APIs/CLIs provided by existing DL frameworks for model conversion make it difficult for developers to correctly choose or use their desired APIs/CLIs; meanwhile, frequent

Example (a) – Core ML issue #525

addition, deprecation, and upgrade of APIs/CLIs also make their usage error-prone.

Repair original model. Repairing the DL model used for conversion fixes 13.6% of faults in *Model Conversion*, which mainly belong to the *Shape/size error (A.3)* and *Unsupported operation (A.6)* categories. As shown in Example (a), the Core ML issue #525 is a real-world example on the *Shape/size error (A.3)*. A developer uses Keras to implement a binary classifier, trains and tests it successfully. However, when she converts the obtained model to the Core ML format, *Shape/size error (A.3)* occurs. Since she specifies two output labels ("0" and "1") during model conversion, the converter expects a model with a two-dimensional output tensor. However, the output of the original model is an one-dimensional tensor, indicating the probability that the input is classified as label "1." To resolve this fault, the developer repairs the original model and makes it output a two-dimensional tensor, with each dimension indicating the probability that the input is classified as one label ("0" or "1"). As for *Unsupported operation (A.6)*, developers often (i) replace it with a supported one, (ii) implement its function outside the model, or (iii) delete it if it is unnecessary.

Fix tensor shape/size specification & **Fix tensor name specification.** The two strategies fix the specification of the shape/size and the name of input/output tensors during model conversion, respectively. As described in previous work [19], training DL models can be expensive since it requires a large quantity of computational resources and labelled data that might not be readily available. Therefore, developers often directly use pre-trained DL models that are available online. In this case, they may have no idea about the model information (e.g., the shape/size and the name of input/output tensors) that needs to be specified during model conversion. Incorrect specification can result in *Shape/size error (A.3)*, *Tensor error (A.2)*, *Unexpected model (A.11)*, etc. Therefore, we can observe that the two strategies mainly fix faults with these symptoms. For example, a developer reuses an object detection model that she is not familiar with from GitHub and specifies the input tensor as a tensor not contained in the model (SO post #55803971), resulting in *Tensor error (A.2)*. The corresponding solution is fixing tensor name specification.

Select TF operator & **Register operator.** The two strategies are used to tackle the *Unsupported operation (A.6)* faults that occur when converting DL models into the TF Lite format. Selecting TF operators allows DL models to use a subset of TF operators that are not supported by TF Lite [41], while registering operators refers to registering unsupported operators in the TF Lite run-time library so that the run-time knows how to map these operators to executable code [42]. Compared to selecting TF operator, registering operator can be used not only to support TF operators, but also to support operators customized by developers.

Change graph type. This group of fixes changes the type of the model graph (e.g., training graph and evaluation graph) used for conversion. The model graph refers to the computational graph that represents the structure of the DL model. Since operations involved in model training and evaluation are not always the same, developers need to construct the training graph and the evaluation graph separately. The graph used for conversion should be the evaluation graph since developers always aim to make inference rather than training on mobile devices. When developers use the training graph for conversion, some training operations may be unrecognized and unsupported by the converter. As a result, developers would encounter *Unsupported operation (A.6)*. *Model parse failure (A.1)* is another common symptom that occurs when the incorrect type of model graph is provided.

Fix/use quantization. This group of fixes selects a proper quantization method according to developers' demand or fixes the incorrect quantization configuration. Naturally, it can resolve the *Quantization failure (A.4)*. In addition, since model quantization can reduce the model size while reducing the precision representations of model weights, when developers observe that the model size does not change as expected after quantization (i.e., *Unexpected model size (A.5)*), there may be a fault in the quantization configuration that needs to be fixed.

Finding 5: *We identify nine frequent fix strategies for faults in model conversion. The three most common strategies are fixing framework installment/version, fixing conversion API/CLI usage, and repairing the original model, resolving 30.6%, 15.6%, and 13.6% of faults in this stage, respectively.*

B. Fix Strategies for Faults in DL Integration

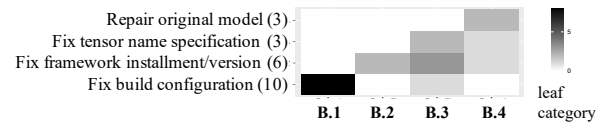As illustrated in Fig. 5, we identify four frequent fix strategies for faults in *DL Integration*.

Fig. 5. Distribution of fix strategies for leaf categories in *DL Integration*.

Fix build configuration. 26.3% of faults in *DL Integration* are resolved by fixing the build configuration of mobile DL projects, including fixing dependency version, fixing link configuration, fixing option settings, etc. This fix strategy mainly resolves the *Dependency resolution error (B.1)*.

The remaining three frequent fix strategies have been described in Section V-A. They are also applicable to some faults in *DL Integration*.

Fix framework installment/version. When the required DL framework is not successfully installed or the DL model is not incompatible with the framework version used in the project, symptoms like *Framework loading failure (B.2)* and *Model loading failure (B.3)* may occur. In such cases, developers need to fix framework installment/version.

Fix tensor name specification. When input/output tensors are specified incorrectly during model conversion, the converted model may not be loaded in mobile projects successfully (i.e., *Model loading failure (B.3)*). Moreover, improper specification of input/output tensors may cause *GPU delegate failure (B.4)*. For instance, a developer encounters this failure since some data pre- and post-processing operators in the original model are not supported by GPU (TF issue #25238). The fixing strategy is re-specifying the input and output tensors during model conversion to ensure that the unsupported operators are not between the new input and output nodes, thereby not in the converted model.

Repair original model. This strategy can resolve the *GPU delegate failure (B.4)*. In fact, some operators supported by DL frameworks are not supported by GPU. In this case, developers can repair the original model to remove these operators and implement alternative operations, so that the integrated DL models can run on the GPU backend of mobile devices.

Finding 6: *We identify four frequent fix strategies for faults in DL integration. The most common one is fixing build configuration, which resolves 26.3% of faults in this stage.*

C. Fix Strategies for Faults in Inference

We identify 13 frequent fix strategies for faults in *Inference* and present the distribution of these strategies in Fig. 6.
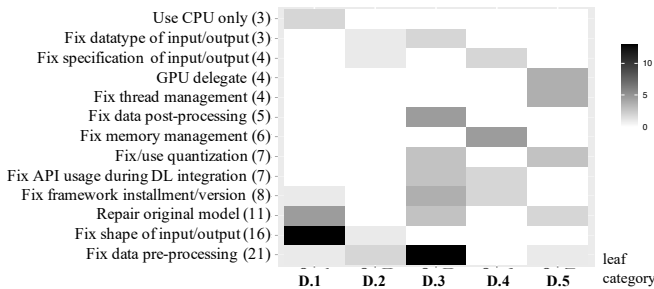

Fig. 6. Distribution of fix strategies for leaf categories in *Inference*.

Fix data pre-processing & **Fix data post-processing.** 23.6% of faults in *Inference* can be resolved by fixing the process of preparing data for model input (i.e., data pre-processing) or the process of parsing model output to obtain expected or human-readable results (i.e., data post-processing). When developing DL models, data pre-processing is often considered as an individual stage [43] and thus may not be included inside the model structure. In this case, code for data pre-processing needs to be re-implemented in the mobile

project during the deployment process, so as to keep the consistent behaviors of the DL model before and after deployment. Forgetting to implement it or implementing it incorrectly can result in unexpected results. In addition, sometimes the model behaves well and generates the expected output, but developers make mistakes in parsing the model output, which can also result in unexpected results. Therefore, we can find that the two fix strategies mainly tackle the *Unexpected result (D.3)* and 48.7% of faults in this category can be resolved by them.

Fix shape of input/output & **Fix datatype of input/output** & **Fix specification of input/output.** When integrating DL models into mobile projects, developers often need to prepare the input/output arrays that are used for storing input/output data and specify their shape and datatype. For example, as shown in Example (b) (SO post #58061111), a developer integrates a DL model with one input tensor and four output tensors into an Android project implemented in Java. First, she uses the model to initialize an interpreter. Then, she allocates memory and specifies the shape and datatype for input and output arrays, and sets these arrays as the model input and output. Finally, she uses input data to fill the input array and make inference. During the above process, when the shape of input/output arrays are incorrectly specified, developers may encounter *Shape/size error (D.1)*. Similarly, when the specification of the datatype of input/output arrays is incorrect, developers may encounter *Datatype/format error (D.2)* or obtain *Unexpected result (D.3)*. Therefore, fixing shape of input/output mainly resolves faults in *D.1*, while fixing datatype of input/output tackles faults in *D.2* and *D.3*. In addition, incorrect specification of input/output of the model may result in faults such as *Datatype/format error (D.2)* and *Memory issue (D.4)*. For example, the symptom of the fault in Example (b) is *Memory issue (D.4)* with an error "*Unexpected failure when preparing tensor allocations*". The corresponding solution is fixing specification of input/output.

Fix API usage during DL integration. In the DL integration process as shown in Example (b), developers often misuse

relevant APIs provided by DL frameworks. The corresponding solution is fixing API usage during DL integration.

Fix memory management. This group of fixes resolves the faults related to memory management during the DL integration stage. A typical fault is that developers may set the input/output arrays before allocating memory for them (e.g., SO post #56819142), which results in *Memory issue (D.4)*.

Fix thread management & **GPU delegate.** The two groups of fixes refer to setting an appropriate number of threads in mobile projects and configuring mobile projects to enable DL models in them to run on the GPU backend, respectively. Both of them can reduce the latency during inference and thus resolve 50% of the faults in *Speed issue (D.5)*.

Use CPU only. This group of fixes forces DL models to run on the CPU backend during inference by configuring some settings in mobile projects. It mainly resolves the *Shape/size error (D.1)*. For example, when a developer makes inference with a Core ML model, an error is thrown, reporting that the size of the input sequence exceeds the upper bound (SO post #52144540). The cause is that a dense operation in the model with a large size of sequences is unable to be performed on a GPU due to the memory constrains. Finally, the developer forces the model to use only CPU and resolves the fault.

In addition, there are three fix strategies that have been elaborated in Sections V-A and V-B.

Repair original model. This strategy mainly resolves the *Shape/size error (D.1)* and *Unexpected result (D.3)*. Specifically, when the input size expected by DL model is inconsistent with the actual size of data extracted in apps (i.e., *Shape/size error (D.1)*), one solution is to reshape the original models. Moreover, some developers find that the models cannot perform well in real applications (i.e., *Unexpected result (D.3)*) and thus choose to refine their original models.

Fix framework installment/version. Fixing framework installment/version can also resolve some faults that occur during inference. For example, a developer gets worse results when she converts a Keras model into the TF Lite format (SO post #51966486). The root cause is that an API that she uses during model conversion is problematic in TF 1.10. Upgrading TF to version 1.11 resolves the fault.

Fix/use quantization. The problematic configuration of model quantization can affect performance of the converted model and thus result in unexpected inference results. Moreover, since the quantized model is more light than the original one, model quantization is also a solution to speed up the inference process. Therefore, fixing/using quantization mainly resolves the *Unexpected result (D.3)* and *Speed issue (D.5)*.

Finding 7: *The fix strategies for faults in inference are diverse. They cover many stages of the deployment process, including fixing data processing, fixing the model conversion stage (e.g., fixing/using quantization), fixing the DL integration stage (e.g., fixing API usage during DL integration), etc.*

VI. Discussion

Given the rapidly increasing popularity of mobile DL apps, our study has timely and immediate implications for developers, especially novice developers. Specifically, our results can help developers more efficiently understand and resolve common deployment faults. For example, a developer may be confused as to how to resolve the *Unsupported Operation (A.6)* symptom, since the fault may lie in the model development process or any setting/configuration in model conversion. However, with our results, the developer can know how such faults are usually resolved in practice so that she can find the solution with less trial and error.

Nevertheless, due to the broad spectrum of deployment faults, it is challenging for developers to detect and fix these faults completely manually. Therefore, we call on SE researchers to develop automated techniques to assist them. Although the combinations of fault symptoms and fix strategies derived in our study can serve as common strategies for the automated techniques, we believe that more research efforts are needed to achieve the goal. Next, we discuss some implications of our findings on future research.

Testing DL models deployed on mobile devices. As suggested in our study, 20.1% of deployment faults (e.g., *Unexpected model (A.11)*, *Unexpected result (D.3)*, and *Speed issue (D.5)*) do not explicitly lead to an error or a crash during deployment, and are thus usually exposed relying on developers' experience or extra efforts. This non-trivial portion indicates the importance of testing deployed models automatically. However, existing testing efforts [44]–[46] are mainly dedicated to the DL models obtained by training, rather than the DL models converted and deployed on mobile devices. Unlike testing the trained DL models, testing deployed DL models on mobile devices has its unique challenges in (i) *resource limitation* and (ii) *undetermined change in model behaviors*. Specifically, compared to the PC/server platforms used for testing trained DL models, mobile devices used for testing deployed models have limited resources in terms of computing power and memory size. In addition, in the cases where quantization techniques are employed during model conversion, the deployed models should have different behaviors from the original models since quantization techniques reduce the precision representations of model weights. However, it is unclear how differently the models after deployment might behave, increasing the difficulty in testing the deployed models. For example, a developer gets worse predictions using a TF Lite model converted from a Keras model (SO post #51966486). Since she employs quantization techniques during model conversion, it is difficult for her to tell whether the performance loss of the model is caused by only the quantization or other bugs in the deployment process. To the best of our knowledge, there is little work focusing on the deployed model testing. With increasing growth of mobile DL apps, we encourage researchers to conduct research in this direction and propose some testing techniques accordingly.

Repairing DL models based on deployment faults. We can find that repairing the original DL models used for deployment is a common fix strategy for faults that occur in model conversion, DL integration, and inference stages. Specifically, it resolves 11.2% of deployment faults, cover-

ing 10 frequent symptoms. Therefore, we believe that this significant fix strategy deserves the attention of researchers. However, existing research efforts [19] focus on repairing DL models in the development process and investigate the correlation between different model repairing patterns and various fault types in the development process, including API faults, data faults, structural faults, etc. By comparison, there is little work on repairing DL models based on faults identified in the deployment process. We call on researchers to develop automated techniques in this direction to facilitate the automated fix of deployment faults of mobile DL apps.

Mining API/CLI usage protocols. In this study, we observe that 34 out of 304 faults are resolved by fixing API/CLI usage in model conversion and DL integration stages. Mining the API/CLI usage protocols enforced by DL frameworks is a promising research topic to facilitate the automated detection and fixing of these faults. Specifically, researchers can mine these protocols from the official documentation of DL frameworks and relevant projects available on open-source code repositories. In particular, the changes in the API/CLI usage protocols caused by the evolution of DL frameworks need to be highlighted in the mining results.

VII. THREATS TO VALIDITY

In this section, we discuss threats to the validity of our study.

Selection of frameworks. Our identification of deployment faults of mobile DL apps is based on two relevant frameworks, which may lead to possible selection bias in this study. To mitigate this threat, we select the representative and widely-used frameworks. On the one hand, the selected frameworks are widely used in industry practice and well adopted in related studies [13], [23]. On the other hand, the selected frameworks cover the deployment scenarios of two typical types of mobile apps (i.e., Android and iOS apps).

Selection of data sources. Since there is no list of all mobile DL app projects in the world, our study cannot cover all the relevant faults, which may lead to a threat to the external validity. To mitigate this threat, we select two representative data sources (i.e., SO and GitHub) that have been widely used in empirical studies in SE [16]–[19], [37]. Since previous studies [18], [47] have found that findings derived from SO and GitHub posts can be well validated by practitioners, we believe that our choice of SO and GitHub does not invalidate our results. However, it is still possible that in other contexts developers may encounter faults that are not covered in this study. In the future, we plan to include interviews with researchers and practitioners to further validate our findings.

Subjectivity of researchers. The subjectivity of researchers presents a possible threat to the validity of manual analysis. To mitigate this threat, we ensure that each case is labelled by at least two authors with an experienced arbitrator resolving the conflicts and inspecting all final results. In addition, the inter-rater agreement is relatively high, which demonstrates the reliability of the labelling schema and procedure.

VIII. RELATED WORK

In this section, we summarize the related work to well position our study within the literature.

Challenges that ML/DL poses for SE. Machine learning (ML) plays an increasingly significant role in various application domains and poses new challenges for software developers [48]. To understand these challenges, Alshangiti et al. [43] analyzed the ML-related questions posted on SO and found that these questions are more difficult to answer than other questions. By further analysis, they demonstrated that model deployment is the most challenging across all the ML phases and that DL-related topics are the most common in the ML-related questions. In recent years, several studies have focused on the challenges in developing DL applications. For example, Han et al. [14] applied an automatic topic modelling technique to the SO questions related to three popular DL frameworks and derived the topics contained in these questions. Their results revealed common concerns that developers face when using DL frameworks, such as version problems and model training. Similarly, Zhang et al. [15] manually analyzed DL-related questions on SO and found that program crashes, model deployment, and implementation related questions are the most frequently asked. Recently, Chen et al. [13] investigated SO questions related to the deployment process of DL based applications. They derived the topics of the specific challenges that developers face when deploying DL models to server, mobile, and browser platforms. In contrast, instead of deriving the topics of challenges at a macro level, we aim to analyze symptoms and fix strategies of the deployment faults and provide actionable implications for fault detection and fix in mobile DL apps. In addition, we do not limit our analysis to just SO and also consider GitHub, which ensures comprehensiveness of this study.

Empirical study on faults. There have been a number of empirical studies that focus on faults in different types of software systems. For example, Lu et al. [49] studied concurrency fault characteristics; Franco et al. [29] explored real-world faults in numerical software; Gao et al. [50] conducted an empirical study on recovery faults in large-scale distributed systems. In recent years, the rapid development of DL technologies has inspired some empirical studies on characterizing the faults in software applications that make use of DL frameworks. For example, Zhang et al. [16] collected faults in TF programs from SO and GitHub. They categorized the symptoms and root causes of these faults through manual analysis. Following this work, Humbatova et al. [18] and Islam et al. [17] extended their scope to the faults in programs written based on five popular DL frameworks to present more comprehensive results. Moreover, Islam et al. [19] analyzed the fix strategies of these faults in their follow-up work. Recently, Zhang et al. [51] studied the program faults of DL jobs running on a remote and shared server platform. Across the existing empirical studies, faults are often characterized based on multiple dimensions, including types, symptoms, root causes, fix strategies, etc. Compared to the prior studies, we

apply these fault characterization methods to the faults in a different domain, i.e, mobile DL apps.

Mobile DL apps. To make DL models accessible for users, developers need to deploy them to different platforms according to various application scenarios. A popular way is to deploy them on mobile devices. To facilitate this deployment process, researchers have proposed many optimization techniques (e.g., cloud offloading [10] and model compression [52]). In addition, researchers have built numerous DL based applications on mobile devices [11], [53], [54]. To bridge the knowledge gap between research and practice, Xu et al. [22] conducted an empirical study on large-scale Android apps collected from Google Play store and demonstrated the increasing popularity of DL in real-world mobile apps. Despite this popularity, the related techniques for deploying DL models to mobile devices are still not very mature. Recently, Guo et al. [23] investigated the performance gap when the trained DL models are migrated from PC to mobile devices with the help of TF Lite and Core ML. Their findings unveiled that the deployment still suffers from compatibility and reliability issues. Despite these efforts, the characteristics of deployment faults of mobile DL apps are still under-investigated and thus we aim to fill in this knowledge gap.

IX. Conclusion

In this paper, we have presented a comprehensive study of deployment faults of mobile DL apps. By manual examination of 304 real-world faults extracted from SO and GitHub, we have derived a taxonomy of fault symptoms with 23 categories, indicating that the process of deploying DL models on mobile devices stretches over a wide spectrum of faults. Moreover, we have analyzed the fixes for the extracted faults and distilled frequent combinations of fault symptoms and fix strategies that can be adopted to facilitate manual and automated fault fix. Finally, we have discussed insightful implications for developers and researchers based on our results.

Acknowledgment

References

[1] Y. Ma, D. Xiang, S. Zheng, D. Tian, and X. Liu, "Moving deep learning into Web browser: how far can we go?" in *Proceedings of the World Wide Web Conference, WWW 2019*, 2019, pp. 1234–1244.

[2] Z. Chen, S. Shen, Z. Hu, X. Lu, Q. Mei, and X. Liu, "Emoji-powered representation learning for cross-lingual sentiment classification," in *Proceedings of the World Wide Web Conference, WWW 2019*, 2019, pp. 251–262.

[3] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "DeepBillboard: systematic physical-world testing of autonomous driving systems," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, pp. 347–358.

[4] Z. Chen, Y. Cao, X. Lu, Q. Mei, and X. Liu, "SEntimoji: an emoji-powered learning approach for sentiment analysis in software engineering," in *Proceedings of the 2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, 2019, pp. 841–852.

[5] Z. Chen, Y. Cao, H. Y. X. Lu, X. Peng, H. Mei, and X. Liu, "Emoji-powered sentiment and emotion detection from software developers' communication data," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 2, pp. 18:1–18:48, 2020.

[6] "An exploration of mobile first AI," https://medium.com/swlh/an-exploration-of-mobile-first-ai-576c944efd36, retrieved on June 27, 2020.

[7] "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/, retrieved on June 27, 2020.

[8] "Artificial intelligence next key growth area for smartphones as numbers top six billion by 2020, IHS Markit says," https://news.ihsmarkit.com/prviewer/release_only/slug/technology-artificial-intelligence-next-key-growth-area-smartphones-numbers-top-six-bi, retrieved on June 27, 2020.

[9] R. J. Wang, X. Li, and C. X. Ling, "Pelee: a real-time object detection system on mobile devices," in *Proceedings of the 2018 Annual Conference on Neural Information Processing Systems, NeurIPS 2018*, 2018, pp. 1963–1972.

[10] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "DeepCache: principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, MobiCom 2018*, 2018, pp. 129–144.

[11] M. Xu, F. Qian, Q. Mei, K. Huang, and X. Liu, "DeepType: on-device deep learning for input personalization service with minimal privacy concern," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, IMWUT*, vol. 2, no. 4, pp. 197:1–197:26, 2018.

[12] M. K. Mustafa, T. Allen, and K. Appiah, "A comparative review of dynamic neural networks and hidden markov model methods for mobile on-device speech recognition," *Neural Computing and Applications*, vol. 31, no. 2, pp. 891–899, 2019.

[13] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *Proceedings of the 2020 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, 2020, pp. 750–762.

[14] J. Han, E. Shihab, Z. Wan, S. Deng, and X. Xia, "What do programmers discuss about deep learning frameworks," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2694–2747, 2020.

[15] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*, 2019, pp. 104–115.

[16] Y. Zhang, Y. Chen, S. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018*, 2018, pp. 129–140.

[17] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, 2019, pp. 510–520.

[18] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, pp. 1110–1121.

[19] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: fix patterns and challenges," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, pp. 1135–1146.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, 2016, pp. 265–283.

[21] "Keras," https://github.com/keras-team/keras, retrieved on June 27, 2020.

[22] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *Proceedings of the World Wide Web Conference, WWW 2019*, 2019, pp. 2125–2136.

[23] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019, pp. 810–822.

[24] "Tensorflow Lite," https://www.tensorflow.org/mobile/tflite, retrieved on June 27, 2020.

[25] "Core ML," https://developer.apple.com/documentation/coreml, retrieved on June 27, 2020.

[26] J. Wang, B. Cao, P. S. Yu, L. Sun, W. Bao, and X. Zhu, "Deep learning towards mobile applications," in *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018*, 2018, pp. 1385–1393.

[27] D. Chen and K. G. Shin, "Turnsmap: enhancing driving safety at intersections with mobile crowdsensing and deep learning," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, IMWUT*, vol. 3, no. 3, pp. 78:1–78:22, 2019.

[28] A. Ferdowsi, U. Challita, and W. Saad, "Deep learning for reliable mobile edge analytics in intelligent transportation systems: an overview," *IEEE Vehicular Technology Magazine*, vol. 14, no. 1, pp. 62–70, 2019.

[29] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, 2017, pp. 509–519.

[30] "Supplemental materials," https://github.com/chenzhenpeng18/icse2021.

[31] "Stack exchange data dump," https://archive.org/details/stackexchange, retrieved on June 7, 2020.

[32] Y. Lou, Z. Chen, Y. Cao, D. Hao, and L. Zhang, "Understanding build issue resolution in practice: symptoms and fix patterns," in *Proceedings of the 2020 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, 2020, pp. 617–628.

[33] "Github search API," https://developer.github.com/v3/search/, retrieved on June 27, 2020.

[34] "Github repository of TensorFlow," https://github.com/tensorflow/tensorflow/, retrieved on June 27, 2020.

[35] "Github repository of Core ML," https://github.com/apple/coremltools/, retrieved on June 27, 2020.

[36] S. Beyer, C. Macho, M. Pinzger, and M. D. Penta, "Automatically classifying posts into question categories on Stack Overflow," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018*, 2018, pp. 211–221.

[37] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 1199–1210.

[38] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[39] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[40] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, pp. 159–174, 1977.

[41] "Select TensorFlow operators to use in TensorFlow Lite," https://www.tensorflow.org/lite/guide/ops_select, retrieved on August 10, 2020.

[42] "Register custom operators in TensorFlow Lite," https://www.tensorflow.org/lite/guide/ops_custom?hl=en, retrieved on August 10, 2020.

[43] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, "Why is developing machine learning applications challenging? A study on Stack Overflow posts," in *Proceedings of 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019*, 2019, pp. 1–11.

[44] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "DeepCT: tomographic combinatorial testing for deep learning systems," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, 2019, pp. 614–618.

[45] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepGauge: multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, 2018, pp. 120–131.

[46] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017*, 2017, pp. 1–18.

[47] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, pp. 590–601.

[48] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, accepted to appear.

[49] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008*, 2008, pp. 329–339.

[50] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, 2018, pp. 539–550.

[51] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, pp. 1159–1170.

[52] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "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 2018*, 2018, pp. 389–400.

[53] V. Radu, N. D. Lane, S. Bhattacharya, C. Mascolo, M. K. Marina, and F. Kawsar, "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 Adjunct 2016*, 2016, pp. 185–188.

[54] G. Mittal, K. B. Yagnik, M. Garg, and N. C. Krishnan, "Spotgarbage: smartphone app to detect garbage using deep learning," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp 2016*, 2016, pp. 940–945.