A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization

Hao Guan[1,2], Ying Xiao[2], Jiaying Li[3], Yepang Liu[2], Guangdong Bai[1]
[1] The University of Queensland, Brisbane, Australia
[2] Southern University of Science and Technology, Shenzhen, China
[3] Microsoft Software Technology Center Asia, Beijing, China
{hao.guan, g.bai}@uq.edu.au, 12150075@mail.sustech.edu.cn, jiayingli@microsoft.com, liuyp1@sustech.edu.cn

***Abstract*—Due to the great advance in machine learning (ML) techniques, numerous ML models are expanding their application domains in recent years. To adapt for resource-constrained platforms such as mobile and Internet of Things (IoT) devices, pre-trained models are often processed to enhance their efficiency and compactness, using *optimization* techniques such as pruning and quantization. Similar to the optimization process in other complex systems, e.g., program compilers and databases, optimizations for ML models can contain bugs, leading to severe consequences such as system crashes and financial loss. While bugs in training, compiling and deployment stages have been extensively studied, there is still a lack of systematic understanding and characterization of *model optimization bugs* (MOBs).**

In this work, we conduct the first empirical study to identify and characterize MOBs. We collect a comprehensive dataset containing 371 MOBs from TensorFlow and PyTorch, the most extensively used open-source ML frameworks, covering the entire development time span of their optimizers (May 2019 to August 2022). We then investigate the collected bugs from various perspectives, including their symptoms, root causes, life cycles, detection and fixes. Our work unveils the *status quo* of MOBs in the wild, and reveals their features on which future detection techniques can be based. Our findings also serve as a warning to the developers and the users of ML frameworks, and an appeal to our research community to enact dedicated countermeasures.

***Index Terms*—Machine Learning, Model Optimization, Bugs**

I. Introduction

In the last decade, machine learning (ML) techniques have received much attention due to their exceptional performance in solving complex problems. They have been applied to a wide spectrum of domains, ranging from optical character and speech recognition [1], [2] to medical diagnosis [3], [4] and autonomous driving systems [5], which are deployed for safety-critical tasks. With their popularization, the scenarios of ML applications are also expanding. They no longer exclusively run on servers with high computational power, but also on various end devices where computation, storage and energy are relatively limited, such as IoT, edge and mobile devices [6], [7].

Deploying ML models on resource-constrained scenarios may encounter challenges though. Contemporary ML models are typically trained with complex structures and a large number of numerical parameters to achieve favorable accuracy and generalization performance. For example, many models for complex tasks such as image processing and natural language processing may contain millions of trainable parameters and hundreds of network layers [8]. Such massive models can be trained on high-performance servers, but often are not readily deployable in production scenarios. They may not perform well or even fail to work on certain deployed devices, as revealed by recent studies [9], [10]. Moreover, when deploying ML models, model consumers need to consider various factors such as service latency and model updates, which are rarely considered during the training phase. Therefore, ML models usually need to be *optimized* in terms of compactness and resource consumption, prior to their deployment.

Model optimization mainly tackles the challenges of the complex structure of ML models and vast amounts of numerical parameters. Various optimization techniques have been proposed so far, and they can be grouped into two main categories, i.e., *pruning* [11]–[14] and *quantization* [15]–[17]. Pruning-based approaches identify and zero out insignificant parameters to reduce the model size, and quantization-based approaches replace parameters of floating-point numbers with lower precision representations to simplify computation. These techniques have been incorporated by the popular ML frameworks, such as TensorFlow [18] and PyTorch [19], and have become the *de facto* pre-deployment model processors.

Similar to the optimization task in other software that handles complex objects, such as in program compilers [20]–[22] and databases [23], ML model optimization is an error-prone process. It may mistakenly result in a defective model that produces different outputs than the original model, consumes excessive prediction time, or even crashes [9], [24]. To generalize, we call these bugs that appear in the optimization phase of ML frameworks *model optimization bugs* (MOBs).

Research efforts have been made to study the reliability of ML frameworks. They mostly focus on the general program bugs in the learning stage [25]–[29], the compiling stage [30] and the deployment stage [9], [31], [32]. Nonetheless, MOBs remain largely unstudied. Researchers have simply treated the optimization as part of the model training or compiling process, despite the great paradigm shifts. For example, the compiling process aims to adapt a model for a particular

Yepang Liu is affiliated with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems. Hao Guan is under the UQ-SUSTech Joint Program. The corresponding authors are Yepang Liu and Guangdong Bai.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00024

platform [30]. It keeps the model's numerical and structural details to preserve model fidelity. In contrast, the optimization process has to manipulate the details of the model.

In this work, we conduct a comprehensive study to understand MOBs. We aim to provide the developers and users of ML frameworks with our findings and insights regarding MOBs, to help them precisely and efficiently pinpoint, if not completely avoid, such an otherwise overlooked category of bugs. For our study, we have collected 371 real-world MOBs from TensorFlow and PyTorch, the most popular ML frameworks, covering every release of their optimizers from May 2019 to August 2022. We review their bug reports, source code, patches, and developer discussions, taking into consideration multi-dimensional characteristics including their prevalence, symptoms, distributions, and life cycle. We also investigate the root causes of the collected MOBs, and summarize the challenges to detect and fix them.

Key Findings. To the best of our knowledge, this is the first study on characterizing MOBs. Our study unveils the landscape of MOBs in popular ML frameworks. Below we summarize our key findings, and we defer more details to Section IV.

- Most MOBs are introduced with the new releases and major revisions of model optimizers in ML frameworks. They often have stayed in the codebases for a long time before they are discovered and reported.
- MOBs often result in subtle and model optimization-specific consequences such as output corruption and accuracy degradation.
- MOBs have distinctive root causes from bugs in other components of ML frameworks, such as *mis-shaping*, *missing support of types/operations* and *metadata errors*, due to the special operations in model optimization.
- Existing approaches are not effective in detecting MOBs due to the complexity caused by hybrid programming languages, diversified platforms, input data constraints, and volatile inference results.

Contributions. This work makes three major contributions.

- We collect a MOB dataset from TensorFlow and PyTorch and conduct the first systematic study on this previously unstudied type of bugs in ML frameworks.
- We reveal the bug patterns of MOBs, and provide insights into their bug-introducing stages, triggering, and oracles to facilitate their detection and localization. Our study can help researchers gain an in-depth understanding of MOBs, and encourage them to enact dedicated countermeasures.
- To facilitate future research on MOBs, including their detection and fixing, we make our dataset publicly available at https://github.com/MOB2022/MOB-dataset.

II. ML Model Optimization

This section introduces model optimization techniques (Section II-A). We also present the general workflow of model optimization, position it in the entire ML life cycle and distinguish it from other procedures like model compiling (Section II-B).

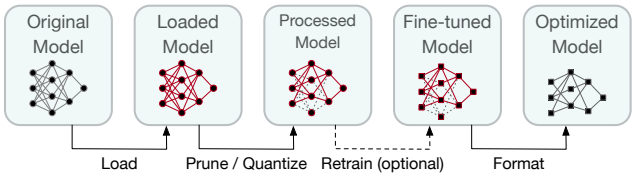


Fig. 1. The general workflow of model optimization

A. ML Model Optimization and Optimization Techniques

Model optimization. The ML models trained to solve real-world complex tasks, especially the deep neural networks, mostly consist of complex structures, and contain a great number of numerical parameters. Using such models may require extensive storage and computational resources, which can easily go beyond the capacity of small devices. Optimization procedures thus are often necessary for adapting the pre-trained models to resource-constrained platforms. Besides being shown beneficial in deploying ML models on resource-restrained devices, model optimization has also demonstrated its advantage in various scenarios, such as reducing latency and cost for inference [11], [33], over-the-air model updates [33], and hardware-specific optimization [34], [35].

Optimization Techniques. Various techniques [11], [15], [33] have been proposed for model optimization. They could be grouped in two genres, i.e., *pruning-based optimization* that alters the network structure, and *quantization-based optimization* that modifies the numerical parameters.

- ***Pruning*** generates sparse models where connections between operators (i.e., neural network layers) are pruned by introducing zeros to the parameter tensors.
- ***Quantization*** represents the models with lower precision, such as 8-bit integers as opposed to 32-bit floats, to simplify computation when using the models.

Over the past few years, these optimization techniques have become mature and effective. They have been realized in the state-of-the-art ML frameworks, and become a *de facto* post-training processing step. For instance, the official guide [33] of TensorFlow introduces how to optimize ML models in the TensorFlow format, and so do PyTorch documents [36], [37].

B. A General Workflow of Model Optimization

Current model optimization processes usually consist of four stages and a typical workflow can be found in Figure 1.

- ***S1: Loading and format conversion.*** The optimizer loads the original model and converts it into an intermediate representation (IR) in a particular format fitting the optimization algorithm. The layers may be annotated with the requirements and configurations for quantization, which can override the default behavior of the original layer.
- ***S2: Pruning/Quantization.*** This is the main step in model optimization. The pruning or quantization operations are applied to the IR to generate an optimized model.
- ***S3: Retraining or fine-tuning*** (optional). The optimized model is fine-tuned or retrained with the original dataset.

This stage is optional, and it aims to ensure the merits of the original model on accuracy are not optimized out.

- ***S4: Output formatting.*** This stage reformats the optimized model from the IR to a model portable for subsequent deployment. All annotations and pre-processing operations should be recovered. Ideally, the optimization should be transparent to the model consumption.

III. Methodology

In this section, we first present our data collection process (Section III-A), and then discuss the methodology for filtering and labeling MOBs (Sections III-B and III-C).

A. Data Collection

We first consider ML platforms from which we construct our MOB dataset. Due to the great popularity of ML techniques in recent years, dozens of frameworks have been developed and publicly released. Selecting representative frameworks is crucial to justify our results and make them generalize to other ML frameworks.

We start with the top-10 ML frameworks among data scientists, according to their power scores [38] synthesized from prevalence-indicating metrics such as usage survey, community activity, and articles/books reference. This list includes TensorFlow (with a power score of 97), Keras (52), PyTorch (23), Caffe (17), Theano (12), MXNet (8), CNTK (5), DeepLearning4j (4), Caffe2 (3), and Chainer (1). Among them, we rule out those that are closed-source, inactively maintained, inadequately documented or having a small number of issues, as of July 31, 2022. This excludes the following ones.

- ***Caffe/Caffe2***. It is excluded as its most recent commit was in early 2020, and its latest stable version was released 5 years ago.
- ***Chainer***. It has not been updated since the end of 2019, and no optimization module is found in it.
- ***Theano***. It is not under active maintenance. The most recent commit was in Nov 2021. Its continued project, *aesara*, is immature without pruning or quantization API.
- ***MXNet, CNTK*** and ***Deeplearning4j***. As of July 2022, their latest version is v1.7, v2.7 and v1.0.0-M2, respectively. They only provide simple quantization [39]–[41], and none of them provides any model pruning API.

This process leaves three winners, i.e. TensorFlow, Keras and PyTorch, and we take them as the representatives of modern ML frameworks. Regarding the actual projects (i.e., repositories), since Keras is built on top of TensorFlow2 and the latest optimization guide of TensorFlow recommends using Keras APIs [42], we treat TensorFlow/Keras as one ML platform in this study. Moreover, the optimization module of TensorFlow resides in a separate repository `tfmot` and thus it is also included in our study. Finally, we take PyTorch, TensorFlow and tfmot as target projects. In Table I, we can see all the projects are large-scale, with millions of lines of code, more than 200,000 commits and more than 30,000 issues in total.

TABLE I
The Statistics of Projects Collected for Studying MOBs

Project Name	Stars	Commits	Lines of Code	Files Num	Issues Num
PyTorch	54,700+	44,566	1,846,722	8,971	24,960[1]
TensorFlow	164,000+	126,552	3,300,942	16,263	6,162
tfmot[2]	1,213	726	30,985	291	103

[1] We counted all issues because no `bug` label is provided.
[2] Tfmot stands for tensorflow/model-optimization, the standalone project for optimizations of TensorFlow/Keras models.

B. Issue Selection

For the selected projects, we leverage their issue tracking systems to collect MOBs. Apparently, not all issues there are related to ML model optimization, and some of them are even not bug reports. Therefore, we take a two-step collection procedure.

Step 1: Selection of MOB related issues. To collect the issues on model optimization, we make use of keywords and labels of each issue. We design several patterns with our desired characteristics and feed them into the GitHub filtering APIs to identify possible MOBs from the massive issues. These patterns are listed in Table II, and they are based on the following two types of rules (both need to be satisfied).

- ***Containing optimization techniques.*** For TensorFlow, we use the label *ModelOptimizationToolkit* in the main repository. For PyTorch, we use the labels *module: pruning* and *oncall: quantization* to filter the issues on pruning and quantization, respectively. For tfmot, we select all issues, since the whole repository is about model optimization.
- ***Containing bug-related labels.*** In TensorFlow, the issue proposers are required to label a bug-related issue with the *bug* label. In PyTorch, the maintainers label valid issues with *triaged* and assign them to relevant developers.

Step 2: Filtering invalid issues. Inevitably, some invalid issues will be accidentally collected in the first step. To filter out such issues that are not qualified for our study, we look into the issue content and apply five exclusion rules (listed in Table III) based on the following principles.

- ***Closed without informative comments.*** These issues do not contain sufficient information for our bug analysis.
- ***Not a bug.*** We exclude those standalone feature requests that are not based on any submitted bugs, project management topics including unit tests or CI/CD, and general questions or enquiries.

With this process, we have collected 371 MOBs, of which 141 are from TensorFlow and 230 are from PyTorch. These bugs are dated from May 2019 to August 2022 and distributed across every release of the optimization modules of each framework. Since we have applied strict selection rules, most of the collected bug reports are well-formed, with clear bug descriptions and actionable code snippets.

C. MOB Labeling

We conduct a manual review on collected issues, focusing on extracting two characteristics of the MOBs, i.e., *symptoms*

TABLE II
SELECTION PATTERNS AND RESULTS FOR MOB RELATED ISSUES

Repository	Filter	Count
pytorch/pytorch	`label:"module: pruning" label:triaged is:issue`	15
pytorch/pytorch	`label:"oncall: quantization" label:triaged is:issue`	321
tensorflow/tensorflow	`label:ModelOptimizationToolkit label:type:bug is:issue`	32
tensorflow/model-optimization	`label:bug is:issue`	127

TABLE III
FILTER PATTERNS AND RESULTS FOR INVALID ISSUES

Rule	Count	Rule	Count
Feature request	31	Documentation	10
CI/CD testing	22	Question / enquiry	6
Closed with no comments	55		

and *root causes*. Since our work is the first on MOB analysis, there are no existing taxonomy criteria to follow. We thus conduct an iterative labeling process based on the open coding methodology [43] to produce a stable and comprehensive taxonomy of the characteristics of MOBs. We explain the labeling iterations below and defer the results to Section IV.

Iteration 1. We study each of the 371 MOBs, and summarize their symptoms and root causes based on the code and discussion in the issues. We take different strategies for issues with and without a patch.

Issues with a patch. The code-level patch provides useful information for us to understand the bug. So we extract the characteristics of patched MOBs from the *pull requests and commits linked to the corresponding issues*. For example, in Tfmot #655, the pull request #607 is referred. We study from the patch in #607 and learn that the problem is about the condition check when modifying layers. Therefore the issue is characterized as a layer operation problem. 115 of the MOBs have a clear patch that exactly fixes the bug.

Issues without a patch. For unresolved issues, we first make sure they are valid based on the rules in Section III-B. Then we derive their characteristics by analyzing the following information.

- ***Error messages in the call stacks.*** For example, the description in Tfmot #367 contains the log of exceptions. We can learn that there is *ValueError* caused by "*Tensor conversion requested dtype float32 for Tensor with dtype float16*", which can be characterized as a type problem.
- ***Input and output from reproduction code.*** For example, PyTorch #80501 has input with large integers, and the output is different as the environment changes. The output can become negative or exactly zero. So we can confidently infer that the issue is about numerical problems.
- ***Comments from framework developers that can explain the problem.*** For example, In PyTorch #76304, the developer replied that "*The issue is that the quantized convolutions are expecting a symmetric padding*". Therefore, we can characterize this issue as a missing supporting problem, which fails to handle a special condition.

While studying the MOBs, we gradually formulate a taxonomy to fit the context of ML model optimization in the following ways: 1) to create subcategories for the types that are too general for MOBs, and 2) to remove the types that are not relevant to MOBs. At the end of this iteration, we manage to compose a preliminary result, which contains two parts: a MOB-specific taxonomy of symptoms and root causes, and the strategy we have adopted to understand the three factors above to decide the category of a MOB.

Iteration 2. Another author is involved in the process of cross-validating the categorization result. This author refers to the preliminary result obtained in ***Iteration 1***, and checks whether the characteristics extracted from the issues conform to the labeling strategies in Section III-C. For the issues that this author does not agree, the two authors discuss them and clarify the criteria used in the strategies. With the updated understanding of the strategies, the second author applies another round of validation to check whether a new disagreement is introduced. Several rounds of discussions have been conducted to reduce the disagreements.

Iteration 3. After the calibration of the categorization, all authors revisit each bug, and agree on every decision regarding the symptom and root cause.

IV. RESULTS AND FINDINGS

This section presents our analysis of MOBs and main findings. We investigate three research questions (RQs).

RQ1: What are the general characteristics of MOBs? This RQ aims to understand the general characteristics of MOBs. We reveal their symptoms, and also present the temporal features of their life cycle, including their trends, time to detection and time to fixing.

RQ2: What are the root causes of MOBs and how are they specific to model optimization? This RQ aims to understand why MOBs occur. We are interested in those atypical root causes that are not captured by existing studies on the bugs of other ML components or life cycle stages.

RQ3: What are main obstacles for detecting MOBs? This RQ focuses on model optimization-specific obstacles that cause challenges for MOB detection.

A. RQ1: General Characteristics of MOBs

We first summarize the characteristics of the MOBs in the wild, focusing on their symptoms and life cycle.

1) Symptoms: MOBs exhibit four types of symptoms: ***Sym1: Crash***. The optimizer exits unexpectedly, and no model is returned.

TABLE IV
THE DISTRIBUTION OF MOB SYMPTOMS

Framework	Crash	OC	MAD	POP	UnDec	Total
PyTorch	149	31	22	19	9	230
TensorFlow	82	34	15	4	6	141
Total	231	65	37	23	15	371

OC: Output Corruption
MAD: Model Accuracy Degradation
POP: Poor Optimization Performance
UnDec: Undecidable type

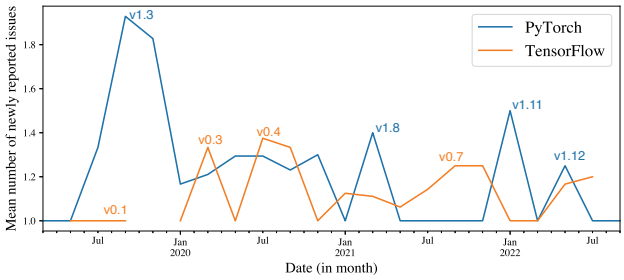


Fig. 2. Trends of MOBs v.s. Versions of PyTorch and TensorFlow

Sym2: Output corruption. The optimizer outputs a malformed model, which becomes incompatible with originally compatible ML programs.

Sym3: Model accuracy degradation. The prediction accuracy of the model after optimization drops severely.

Sym4: Poor optimization performance. The optimizer consumes excessive computational resources (e.g., CPU and storage) or takes abnormally long to complete.

Undecidable type. The symptom is not described clearly in the issue, without the expected and actual behavior.

Table IV shows the distribution of the symptoms of our studied MOBs. Crash (***Sym1***) is the most reported type; 61.3% of MOBs in PyTorch and 65.7% in TensorFlow produce a crash. This may be because this symptom is easily observable by the developers and users. Apart from the crash, 31% of MOBs in PyTorch and 23% in TensorFlow lead to output corruption (***Sym2***). Most of them produce incorrect models that lead to errors when their downstream consumers load them. Therefore, they are also noticeable to users, and the reported issues often contain detailed bug descriptions. The remaining two types of symptoms (***Sym3*** and ***Sym4***) are rare. Only 2.5% in PyTorch and 2.2% in TensorFlow are related to model accuracy degradation (***Sym3***), and the poor optimization performance (***Sym4***) MOBs account for 5% in PyTorch and 2.2% in TensorFlow. A possible reason is that these two types of symptoms are often subtle and users cannot confidently determine whether the degradation and poor performance are caused by bugs or other factors.

2) Life cycles: We investigate the temporal features of MOBs, to reveal their life cycles from three aspects, trends in the development process, time to detection, and time to fix.

Trends of MOBs. Figure 2 presents the time points when the MOB-related issues are submitted on GitHub, and how their fluctuation relates to the major changes in each framework.

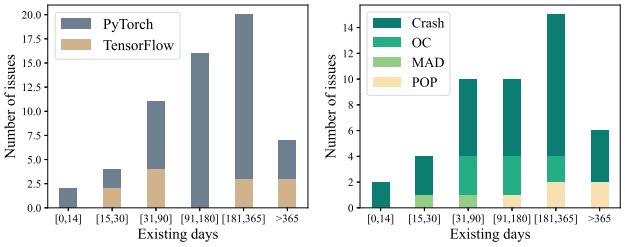


Fig. 3. Time to detection of MOBs grouped by frameworks and symptoms

For PyTorch, the first increase occurred in 2019 Q4, because the v1.3 released on 11 Oct 2019 introduced the quantization APIs for the first time. The next surge happened on 5 Mar 2021, when the v1.8 changed the signature of quantization APIs, and the way of quantization for different layers, including *relu* and *sigmoid*. The v1.11 released on 11 Mar 2022, which introduced new quantization operations and pruning strategies, also caused another significant increase.

A similar phenomenon is observed in TensorFlow. The increases of MOB reports align to new versions and features in the model optimization module of TensorFlow, including v0.3 (initial release of the Keras quantization API), v0.4 (support more convolutional and dense layers that are commonly used in Keras) and v0.7 (switch to new default wrapper).

Time period to detection. For each MOB, we figure out two critical time points: (1) when the MOB is introduced and (2) when it is reported in an issue. As there is no clear detection time, we take the report time of a MOB as the estimated *time point of detection*, on the assumption that developers/users usually report new bugs shortly after they detect them. To find the time point when the MOB is introduced, we study its relevant pull request and/or commits that fix it. We then use *git blame* to track the historical commits and locate the commit that introduces the bug. We use the time of this commit to indicate the *time point of introduction*.

We have found 115 MOBs with a clear fixing commit, and found the time point of introduction for all of them. Figure 3 presents our results, with breakdowns in terms of frameworks and symptoms. Most MOBs have been existing for a long time (>30 days) until they are detected. Among them, the MOBs of poor optimization performance (***Sym4***) generally take a longer time (over 90 days and even 1 year) to be exposed.

Time to fix. We use the time of the pull request or commit that fixes the MOB as the *time point of fix*, and use the time period between it and the time point of detection to estimate the time period to fix the MOB. Figure 4 shows the results. The time period to fix turns out to be extreme for fixed and unresolved MOBs. Most fixed MOBs take short time to be resolved, while the unresolved issues are usually kept open for a long time. We study the code and discussion in each issue to figure out the reason for this difference. We find that the fixing of issues depends on the difficulty level of reproduction, which can be influenced by the following factors.

- ***Non-crashing bugs***. Bugs of ***Sym2***, ***Sym3*** and ***Sym4*** are

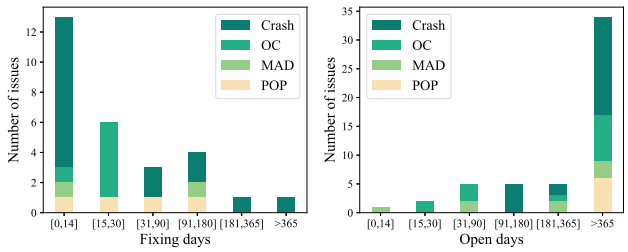


Fig. 4. Fixing status of MOBs grouped by symptoms

harder to reproduce than crash (***Sym1***), because the output of ML programs is stochastic in nature.

- ***Complexity of models.*** Buggy functions for a single tensor are more likely to be fixed, while the problematic optimizations for a complete model is usually too complex to address, requiring more efforts of developers.

Answer to RQ1: The reported MOBs mainly cause four types of symptoms, including *crash*, *output corruption*, *accuracy degradation* and *poor performance*, where *crash* accounts for a majority. MOBs are mostly introduced during the first introduction or the major updates on the optimizers of the ML frameworks. Most MOBs are hidden in the codebases for a long time (>90 days) before they are discovered and reported.

B. RQ2: Root Causes of MOBs

In this section, we present the root causes of MOBs, and discuss their uniqueness.

1) Root causes: Our labeling process (Section III-C) categorizes the root causes of MOBs into five different types. Below, we provide their definitions, analyze their consequences, and present typical MOBs as illustrative examples.

RC1: Wrong type (89). The MOBs are triggered when an input of a wrong type is passed into the optimizer. Such a case happens when an input model of unsupported format is fed into the optimizer or a function makes wrong assumptions of input parameter types, especially when the inputs are of collection types. We observe 89 such cases.

Consequence. As shown in Figure 1, the model optimizer first loads the input model and converts it to an intermediate representation (IR). When the optimizer fails to support or makes wrong assumptions on the format of the input model, an exception of *TypeError*, *AttributeError* or *ValueError* could be triggered, which may crash the program. This type of MOBs can also occur when the optimizer processes the IR, which may consist of various collections like lists or dictionaries, but assumes all elements of the IR are of the same type.

Illustrative example. We show an MOB of tfmot #753 in Figure 5. After **S1** stage, the function `collect_prunable_layers` aims to select layers for pruning. However, the input *model* may contain a collection of models (*submodules*) instead of layers. The original code assumes that all elements are of the layer type and thus fails

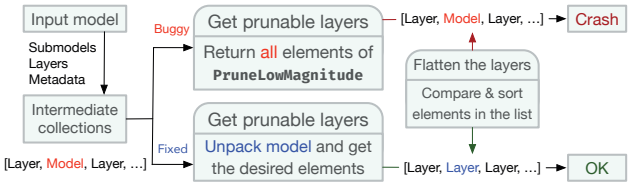


Fig. 5. Data/control flow diagram showing the MOB tfmot #753 and its fix

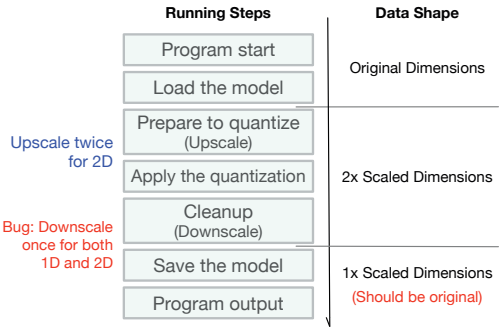


Fig. 6. Data flow of PyTorch #59200

to handle such a case. Next, the output *prunable_layers* are passed to a sorting step, where comparisons among these *layers* are performed. Since the comparisons between layers and models are not implemented, this step can be invalid and leads to a crash. To fix this issue, the developer has to add a type check and process different types in separate branches.

RC2: Unexpected shapes (41). The optimizers operate frequently on tensors. Improper transformations or implementations can easily produce malformed tensors of unexpected shapes, which cannot be digested by further operations.

Consequence. Such a MOB often raises an exception of *IndexError*, *KeyError* or *AssertionError*. This may cause the program to crash or corrupt the output.

Illustrative example. Figure 6 demonstrates a typical example of shape handling in PyTorch #59200. During batch normalization, the optimizer needs to create two fake dimensions for a 2D input and remove them after processing. However, the original code of PyTorch only removes one, and as a result, the structure of the layer breaks. With such layers, the further computation could produce wrong results or cause a crash.

RC3: Missing supporting data types (31). The data structures used by the optimizers may contain various attributes. The MOBs may appear when some attributes are handled incorrectly, although the data type is claimed to be supported.

Consequence. The issue can occur in the I/O stage or the processing stage. In the I/O stage, such an issue is triggered by some controlling flags during the conversion step. In the processing stage, such an issue is raised during computation, since the precision is not supported and thus the operation cannot be completed. Such MOBs usually cause exceptions of *TypeError* or *AttributeError*.

Illustrative examples. Since PyTorch did not support some

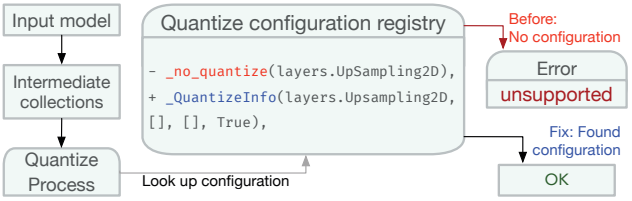


Fig. 7. Diagram of tfmot #372

specific precision of floating points in its early versions, several issues, such as #42351 and #32775, which are related to FP16 types, were raised by users. To mitigate the problem, a pull request #52612 is created to track the status of FP16 support.

For controlling flags related bugs, `TFOpLambda` is a special logic layer with the following flag to guard metadata: `_preserve_input_structure_in_config`. But, during the quantization step, this flag prevents the adaption of the format and leads to a *TypeError* exception. This is due to unsupported handler for this type of layer (shown below in Listing 1 and 2).

```
# Preserve all argument data structures
# when saving/loading a config
self._preserve_input_structure_in_config = True
```

Listing 1. Special Flag for the `TFOPLambda` Layer

```
if input_tensors in not None:
    if not layer._preserve_input_structure_in_config:
        input_tensors = (
        # A necessary operation without which a
        # type mismatching will occur later
            ...
    output_tensors = ...
```

Listing 2. Quantization code fails to handle special flag correctly

RC4: Missing supporting layer operations (109). During model optimization, some layers might need special handling mechanisms, which can be missed by the implementation.

Consequence. The program clearly reports unsupported layers and raises a runtime exception.

Illustrative example. We take tfmot #372 as an illustrative example. As shown in Figure 7, before quantizing a layer, TensorFlow searches for its configuration. However, such configurations might be undefined in the configuration registry, which leads to an unsupported error. In fact, the compatibility of all kinds of layers is not mentioned in the documentation. As a result, there is little chance for the users to know that their design of networks is not suitable for quantization until they run the code. After the quantization of this type of layer (i.e., `Upsampling2D`) is supported by the framework, the configuration is added to the registry so that the program can find it and apply further processes successfully.

RC5: Metadata conversion errors (22). MOBs can occur when the optimizer loses or mistakenly changes the metadata of the input model.

Consequence. Such an issue usually occurs when the ML model is loaded or saved. For example, the names of original

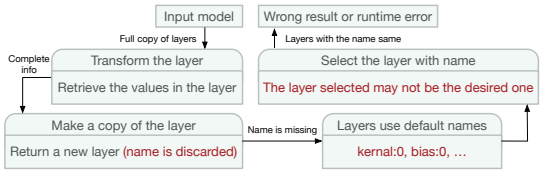


Fig. 8. Diagram of tfmot #317

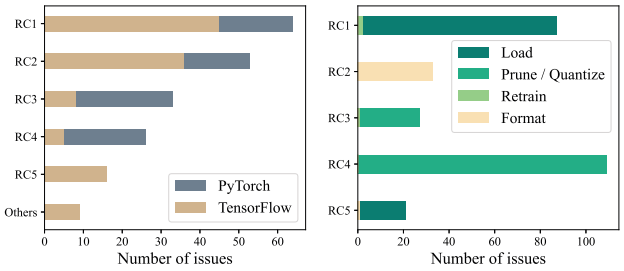


Fig. 9. Root Cause Distribution among Optimization Bugs

model layers are not encoded and stored properly during **S1** stage. As a result, in the **S4** stage, the optimizer is not able to recover and produce the correct name for the output, which may cause the model to behave unexpectedly, because the application may select the layers with their names. In addition, some information of the layer is not changed after conversion. For example, the name and shape of the original layers should be encoded for the intermediate model. The encoded name is important for quantization or pruning. The layers that are not encoded correctly may cause crashes or produce wrong results.

Illustrative examples. Issue #889 in tfmot is a typical case of data loss. In the code, the data type of the layer is assigned manually by the developer. The conversion encodes the layer name with the new data type and also decodes the name with it. As a result, the output has a different layer from the input.

Issue #317 in tfmot shows an example of wrong metadata conversion. The process is shown in Figure 8. The code aims to encode the name of the weights in the layers. However, it fails to handle the sub-classed layer. As a result, the weights from a regular layer and sub-classed layer have the same name, which will cause an error of duplicated keys, resulting in wrong values of weights or exceptions.

Others. Besides the above five categories of MOBs, 12 cannot be categorized into any category. Their root causes are diverse, including legacy or opaque issues.

2) Statistics of root causes: We summarize the overall statistics in Figure 9. Note that 45 MOBs are excluded from this figure as their root causes cannot be determined due to non-reproducible or unresolved issues. In general, the missing supporting data types (**RC3**, 31) and layer operations (**RC4**, 109) are the major root causes of the MOBs. They account for 49.3% in total. The wrong type (**RC1**, 89) is the second common root cause. We also classify these root causes based on the optimization stages (see Section II-B), and the statistics are shown in Figure 9.

3) Comparison with other bugs in ML frameworks: MOBs demonstrate their uniqueness compared with bugs in other components or life cycle stages of modern ML frameworks. The wrong type bugs are found to be the most frequent bugs in other components [27], [30], [31], [44], [45], including model compiling [30]. This may be because Python uses a dynamic typing system, and its grammar offers much flexibility. In fact, the problem can occur in most Python programs, as it can be introduced by almost every part of the code, such as the assignments of literal values, the orders of arguments passed to methods, and the elements in a list or dictionary. In contrast, MOBs are mostly caused by special operations on models or development processes in the context of model optimization. First, model optimization entails a large number of operations that change data values and types. For example, quantization changes a float to an integer of quantized data type (e.g., `quint8`), which is different from Python's built-in integer type. This may cause the missing supporting data types (**RC3**) and layers operations (**RC4**) Second, model optimization changes the model structure by pruning, and changes the metadata including layer type by quantization. This may cause the unexpected shapes (**RC2**) and metadata conversion error (**RC5**). Third, ML frameworks evolve fast, and introduce various new layers and model types. The optimization techniques may fail to keep the same pace, causing the missing supporting data types (**RC3**) and layer operations (**RC4**).

Answer to RQ2: Five main root causes of MOBs are identified. The majority of MOBs arise with model optimization-specific operations and processes distinctive from other components of ML frameworks. This raises the necessity of future studies specifically focusing on MOBs to facilitate their detection and fixing.

C. RQ3: Challenges of MOB Detection

Our study in RQ1 (Section IV-A) reveals that more than 70% of patched MOBs take over three months to get discovered and reported. Therefore, we investigate the reasons why MOBs are hard to detect in this research question.

We review existing approaches to detecting bugs from ML frameworks, and identify the factors their detection techniques rely on, as summarized in Table V. We then assess the feasibility to obtain them from model optimizers, and we have figured out four obstacles listed in Table VI.

1) Obstacles: Below are the four identified obstacles.

Ob1: Hybrid programming languages. The model optimizers of TensorFlow and PyTorch are both implemented in a hybrid way using C++ and Python.

Ob2: Diversified hardware and platforms. The data representation, value ranges, and data operations may have huge differences across hardware and platforms.

Ob3: Large input data with complex constraints. An ML model typically consists of a huge amount of data, and many constraints on the types and shapes should be satisfied.

TABLE V
EXISTING APPROACHES TO DETECTING BUGS IN ML FRAMEWORK

Approach	Factors	Techniques to obtain factors
Static analysis	F1: Call graph	Abstract syntax tree [46]–[48]
	F2: Data abstraction	Pointer analysis [46]
		Tensor partitioning [47]
		Suspect loss estimation [48]
Testing	F3: Input generation	Model-based [48]–[50]
	F4: Test oracles	Class-based distance [49]
		Mean absolute deviation [49]
	F5: Mutation strategies	Genetic algorithms [50]
		Gradient back-propagation [48]

TABLE VI
FACTORS THAT MAY BE IMPACTED BY OBSTACLES IN OPTIMIZERS

Obstacles	Affected Factors	# MOBs
Ob1: Hybrid PL	F1: Call graph	20
	F2: Data abstraction	
Ob2: Hardware and platform	F2: Data abstraction	12
	F4: Test oracles	
Ob3: Data constraints	F3: Input generation	42
	F5: Mutation strategies	
Ob4: Volatile results	F4: Test oracles	9
	F5: Mutation strategies	

Ob4: Volatility of ML models. The weight values and layer structures of ML models are subtle and sensitive. Altering them may cause significant changes in the model performance like accuracy and fidelity.

2) Challenges for static analysis: Static techniques analyze programs based on their control flow or data flow. Gathering either type of information is non-trivial in model optimizers.

F1: Call graph construction. Call graph (CG) construction is an essential and prerequisite step to conducting inter-procedural static analysis. CG construction for hybrid programs remains challenging nowadays, despite some recent advances [51]. Due to **Ob1**, it can be difficult to precisely link the callee function on the C++ side to the call site on the Python side, since identifying the calling relationship may require non-static information, such as type and value information of input parameters, as well as the configuration, which is constructed dynamically by Python.

Illustrative examples. PyTorch #58055 is a concurrency bug caused by a data race in the low-level C++ code. The Python code invokes function `quantize_per_tensor`, whose implementation is located at the C++ side. PyTorch runtime identifies the actual callee function through an internal module named *torchgen*, which dynamically addresses native functions. Such dynamic behavior and internal mechanism of run-time make existing static analyzers fail to extract the precise calling relationship among code in different languages and, as a result, hinder them from detecting this bug.

F2: Data abstraction. To analyze the data flow, static analyzers usually construct an abstract representation of the data for their types and values. First, hybrid languages (**Ob1**) can obstruct the process of data abstraction. Pointer analysis [46] requires the information of all functions that reference the

data. The track of data in Python can easily get lost when coming to C++. Diversified hardware and platforms (**Ob2**) also affect the correctness of data abstraction. For example, tensor partitioning [47] techniques are based on value analysis. However, possible values, especially integers, may be hardware- or platform-dependent, which can be largely unknown during static analysis. Existing studies listed in Table V do not consider how the data is represented in native code, so they may fail to detect cross-platform bugs in optimizers.

Illustrative examples. PyTorch #60077 is an integer overflow bug reproducible on ARM platforms. Yet, the code runs correctly on x86 platforms. This is due to the width distinction of integer variables between the two platforms. Specifically, an integer occupies 64 bits on x86 while 32 bits on ARM, and thus an input value of 2^{32} will trigger a bug only on ARM platforms.

3) Challenges for dynamic testing: Dynamic testing generates test cases to expose potential bugs. The test cases generated by existing techniques may not be valid, or effective in testing model optimizers.

F3: Input generation. Diverse test cases are necessary to cover possible occasions and detect the ones that can cause a bug. Existing techniques [50], [52] generate inputs by mutating seed models. However, they cannot be directly applied to expose MOBs in model optimizers. Due to **Ob3**, mutating model layers is a non-trivial task. Model layers and weights usually have a number of constraints, which are specified explicitly or implicitly, to be satisfied. Even a tiny mistake could invalidate the resulting ML model. Besides the internal constraints imposed by a certain layer, mutation methods also need to consider the layer's external constraints posed by other layers, thus complicating the whole mutation procedure.

Illustrative examples. The models reported in tfmot #753 and PyTorch #67030 contain a special type of layer called *sub-model*, which is different from other layers in terms of layer construction and usage. It requires the test generator to be aware of such special layers and specific mutation operators to transform them. Otherwise, the generated test cases will always miss the layer and fail to detect its bug.

PyTorch #63234 and #63356 are bugs due to incorrect handling of data binding between Python and C++. The test cases for such MOBs are difficult to construct because the data in Python are of dynamic types, and models contain complex structures and attributes. Some attributes are stored in a dictionary, and the possible keys and values are not defined explicitly in the code.

F4: Test oracles. To test ML frameworks, existing dynamic techniques need to construct test oracles. The bugs are exposed by executing the frameworks and checking if some pre-defined assertions are violated. Different from crashing bugs that can be easily observed, non-crashing bugs are often neglected by developers and users, as confirming them is a non-trivial task. Towards this problem, an existing study [49] in detecting ML framework bugs proposes to compare inference results. However, such a technique is not applicable in detecting MOBs mainly because of **Ob4**. The output of model optimization is another model whose structure and weights may greatly differ from the input model. **Ob2** is also a severe problem for constructing test oracles, because the behaviors on different platforms may be naturally different. So far, there exist no effective criteria to facilitate validating such inconsistencies.

Illustrative examples. Tfmot #722, #450 and PyTorch #29024 report bugs about the optimization results. However, when users suspect the correctness of the optimizer, it is not easy to provide solid evidence to support their arguments. Sometimes the framework maintainers cannot even track the issues, and as a result, these issues remain open as of March 2022. On the other hand, tfmot #599, #439 and PyTorch #46180 present some goals that should be optimized, such as the size and the prediction efficiency. These problems are also hard to address, because the pre-set goals may not be reachable at all. The developers and users had a long discussion and finally agreed that these are non-bug issues. The unsatisfactory outputs are due to the inherent limitation of the optimization algorithm.

Tfmot #635 shows different behaviors on CPU and GPU, where CPU returns 0 but GPU returns a tiny floating point number. Similarly, Tfmot #771, PyTorch #58130, #41115 and #36802 are all specific to CUDA GPU acceleration. These cases are using different setups, including graphic card models, CUDA driver versions and builds of frameworks. According to the developers' discussion, framework maintainers even cannot reproduce the issues due to the lack of specific hardware.

F5: Mutation strategies. For dynamic testing, the mutation strategies are applied to analyze the feedback of previous test cases and guide the generation of new test cases. Existing studies [48], [50] use search-based algorithms to guide the test case generation and make the input more likely to trigger bugs. For MOBs, such searching strategies may not be effective.

First, due to **Ob3**, the mutated variants generated by current algorithms can be invalid. The algorithms are designed to mainly mutate weights and external values. They do not consider the constraints of layer structures. For valid variants generated by such algorithms, the layer structures are often not mutated sufficiently, so the variants are not effective to expose MOBs. For the variants that mutate layer structures significantly, very few of them are valid for testing.

Second, **Ob4** makes it hard to calculate a target for guiding the generation, because there are no suitable test oracles to help judge whether a model optimizer behaves properly or not. Current algorithms usually rely on the comparison of inference results, invalid numbers (i.e., NaN, infinity) and crashes. As discussed earlier, the comparison criteria for the results of model optimizers are still missing. The quantization technique of model optimization is involved with integers, which has exclusive problems on overflow (PyTorch #60077), but the current algorithms for invalid numbers mainly deal with floating point numbers.

Answer to RQ3: Four model optimization-specific obstacles can cause prominent challenges in detecting MOBs.

Existing approaches are either inapplicable because of the hybrid programming languages and diversified platforms, or ineffective because of complex data constraints and volatile outputs.

V. Discussion

A. Implications

Based on our bug characterization and the analysis of challenges, we provide several suggestions for developers and users, in order to help improve the model optimization modules in ML frameworks.

Towards Correct Usage. Users should realize that the model optimization modifies ML models in multiple aspects apart from the weights. The modification of data type and layer structure may influence the usage of optimized models, such as the methods of saving and loading models.

Towards Rigorous Development. First, it is recommended to introduce sufficient regression tests to validate the newly supported layers or configurations before a new version is released. Second, developers should pay attention to the side effects of an optimization operation, and identify whether to introduce, remove, or keep the metadata.

Towards Effective Detection. First, it is necessary to construct a clear function mapping between the code in different programming languages, so that static analysis tools can figure out the low-level code in depth and find potential bugs. Second, test cases for detecting MOBs should cover different kinds of layers sufficiently.

B. Feedback from the Community

To validate our findings, we have submitted issue comments, pull requests, and email surveys to the community. Some of our analyses on the operation bugs are confirmed as a worthwhile problem (e.g., tfmot #867). Our pull requests for preventing shape errors in quantization are also accepted and merged (e.g., pytorch #81547).

C. Limitations and Threats to Validity

Our study focuses on understanding the landscape of MOBs in popular ML frameworks. To the best of our knowledge, this is the first comprehensive study on MOBs. However, as the first attempt in this area, our study has several limitations that we target to address in our future work.

First, our study selects extensively used open-source ML frameworks and collects MOBs from their code repositories. Although the collected data are representative, there is a chance that MOBs in other ML frameworks show different characteristics. Second, as there is no existing study on MOBs, we have to design our own selection criteria to identify MOBs. Because the number of issues is huge, our current criteria contain some strict conditions (e.g., issues should have specific labels) to avoid retrieving too many invalid issues, but we may miss real MOBs. In the future, researchers may further explore how to formulate better criteria to collect MOBs. Third, we have to resort to manual efforts to propose labels and patterns for investigating MOB symptoms and root causes. This may be unavoidable as the first study in this area. Although we have checked our taxonomy across the authors based on the open coding methodology to reduce random errors from one person, biases or mistakes may still exist. Fourth, although our data collection has covered the entire development process of the model optimizers, the obtained dataset is still relatively small. Future work could consider extending our dataset by incorporating other data sources, such as the developer forums of ML frameworks, and Q&A websites like StackOverflow.

D. Comparison with Optimization Step in Model Compiling

Deep learning model compiling has a step called *optimization* [53], which is often confused with model optimization. It is worth discussing the fundamental differences between these two. We refer the reader to a recent empirical study [30] for bugs in model compiling.

Targets. Model compiling mainly aims to compile a model to be compatible with a particular platform, while model optimization aims to optimize a model's complexity and performance. Model optimization has to alter the numerical or structural details of models, so it is more subject to new bugs like mis-shaping (**RC3**) and unsupported types (**RC4**).

Strategies. Model compiling typically adopts conservative strategies, e.g., zero-dim-tensor elimination, to preserve the model fidelity. Model optimization is more aggressive, e.g., manipulating non-zero parameters, so it is more likely to lead to type and shape errors that rarely occur in model compiling.

VI. Related Work

Our work explores one type of deep learning bugs, so we present a review of existing work on deep learning bugs.

Bugs in deep learning frameworks. A line of research focuses on the general software bugs in deep learning frameworks. Jia et al. [44] analyze the symptoms and root causes of 202 bugs in TensorFlow. Chen et al. [45] conduct a more comprehensive study of the DL framework bugs in PyTorch, MXNet and DL4J. They analyze the bugs with more aspects, including the user-level APIs, graph-level implementation, operation implementation, general utility and environment-dependent processing.

In addition to the general bugs of deep learning programs, many recent studies focus on specific types of bugs in deep learning programs. Chen et al. [31] explore the faults when deploying deep learning applications on mobile devices. Cao et al. [54] present the characterization of performance bugs that can slow down the execution of DL systems. Zhang et al. [47] target the numerical bugs that occur in DL programs and propose an approach of static analysis on tensor partitioning and affine relation to detect them. Yan et al. [48] follow this work and manage to expose the numerical bugs with dynamic techniques, based on gradient back-propagation. Shen et al. [30] conduct a study on the taxonomy of symptoms and root causes of bugs in DL compilers, which converts the input models into low-level programs for different hardware to run.

Bugs in use of deep learning frameworks. As deep learning has been extensively used in various domains, some studies explore the bugs in the software using deep learning frameworks. Zhang et al. [27] mainly focus on the life cycle of a TensorFlow-based software. They collect 175 real-world bugs from GitHub and StackOverflow. By analyzing the symptoms and root causes of these bugs, they reveal the challenges in bug detection and localization. Humbatova et al. [25] expand their study to Keras and PyTorch. They select 564 projects that use the three frameworks and collect 375 bugs to study. Their taxonomies are characterized by structured interviews with researchers and practitioners, and this work confirms that the bugs could be experienced in practice. Islam et al. [26] further expand the study to Caffe and Theano. They analyze 970 collected bugs for the root causes, and reveal common anti-patterns when using the frameworks.

VII. Conclusion

In this paper, we present the first systematic study of machine learning model optimization bugs by analyzing 371 issues retrieved from the two most popular machine learning frameworks on GitHub (i.e., TensorFlow and PyTorch). We categorize these bugs based on their symptoms and root causes, and summarize the challenges in detecting them. Based on our findings, we also propose suggestions for practitioners to help them detect and avoid model optimization bugs when developing or maintaining machine learning frameworks.

Acknowledgment

We thank our anonymous reviewers for their constructive comments and suggestions on the paper. This work is partially supported by the National Key Research and Development Program of China under Grant No. 2019YFE0198100 and the National Natural Science Foundation of China under Grant No. 61932021, and the University of Queensland under the UQ NSRSG grant and the Global Strategy and Partnerships Seed Funding.

References

[1] M. Namysl and I. Konya, "Efficient, lexicon-free ocr using deep learning," in *2019 international conference on document analysis and recognition (ICDAR)*. IEEE, 2019, pp. 295–301.

[2] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: an overview," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 8599–8603.

[3] M. Bakator and D. Radosav, "Deep learning and medical diagnosis: A review of literature," *Multimodal Technologies and Interaction*, vol. 2, no. 3, p. 47, 2018.

[4] G. Litjens, C. I. Sánchez, N. Timofeeva, M. Hermsen, I. Nagtegaal, I. Kovacs, C. Hulsbergen-Van De Kaa, P. Bult, B. Van Ginneken, and J. Van Der Laak, "Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis," *Scientific reports*, vol. 6, no. 1, pp. 1–11, 2016.

[5] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, "Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1025–1032.

[6] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[7] X. Dai, I. Spasić, B. Meyer, S. Chapman, and F. Andres, "Machine learning on mobile: An on-device inference app for skin cancer detection," in *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2019, pp. 301–305.

[8] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 843–852.

[9] 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 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 750–762.

[10] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 810–822.

[11] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[12] M. H. Meng, G. Bai, S. G. Teo, and J. S. Dong, "Supervised robustness-preserving data-free neural network pruning," 2022. [Online]. Available: https://arxiv.org/abs/2204.00783

[13] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5687–5695.

[14] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5058–5066.

[15] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[16] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[17] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," in *International Conference on Learning Representations*, 2018.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: A system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems (NeurIPS)*, vol. 32, 2019.

[20] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[21] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[22] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 849–863.

[23] M. Rigger and Z. Su, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1140–1152.

[24] GitHub.com, "Issues · tensorflow/model-optimization," Mar 2022. [Online]. Available: https://github.com/tensorflow/model-optimization/issues/

[25] 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 ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, p. 1110–1121.

[26] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, p. 510–520.

[27] Y. Zhang, Y. Chen, S.-C. 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, pp. 129–140.

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

[29] Y. Xiong, Y. Tian, Y. Liu, and S. Cheung, "Towards actionable testing of deep learning models," *SCIENCE CHINA Information Sciences*, 2022. [Online]. Available: https://www.sciengine.com/SCIS/doi/10.1007/s11432-022-3580-5

[30] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 968–980.

[31] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 674–685.

[32] A. Makhshari and A. Mesbah, "Iot bugs and development challenges," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 460–472.

[33] TensorFlow, "Tensorflow model optimization," Mar 2022. [Online]. Available: https://www.tensorflow.org/model_optimization/guide

[34] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2017, pp. 1–8.

[35] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2016.

[36] PyTorch, "Quantization," 2019. [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[37] ——, "Pruning tutorial," 2021. [Online]. Available: https://pytorch.org/tutorials/intermediate/pruning_tutorial.html

[38] J. Hale, "Deep learning framework power scores 2018," Nov 2018. [Online]. Available: https://www.kaggle.com/discdiver/deep-learning-framework-power-scores-2018

[39] Microsoft, "Cntk network optimizations," Jan 2018. [Online]. Available: https://github.com/microsoft/CNTK/blob/v2.7/Manual/Manual_How_to_use_network_optimizations.ipynb

[40] "Mxnet python api." [Online]. Available: https://mxnet.apache.org/versions/1.7/api/python/docs/api/

[41] "Eclipse deeplearning4j." [Online]. Available: https://deeplearning4j.konduit.ai/

[42] Keras, "Keras documentation: The keras ecosystem," 2022. [Online]. Available: https://keras.io/getting_started/ecosystem/

[43] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 120–131.

[44] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *International Conference on Database Systems for Advanced Applications*. Springer, 2020, pp. 604–620.

[45] J. Chen, Y. Liang, Q. Shen, and J. Jiang, "Toward understanding deep learning framework bugs," *arXiv preprint arXiv:2203.04026*, 2022.

[46] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: Analysis for machine learning programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. Association for Computing Machinery, 2018, p. 1–10.

[47] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 826–837.

[48] M. Yan, J. Chen, X. Zhang, L. Tan, G. Wang, and Z. Wang, "Exposing numerical bugs in deep learning via gradient back-propagation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 627–638.

[49] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[50] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* IEEE, 2020, pp. 486–498.

[51] A. M. Bogar, D. M. Lyons, and D. Baird, "Lightweight call-graph construction for multilingual software analysis," *arXiv preprint arXiv:1808.01213*, 2018.

[52] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* 2020, p. 788–799.

[53] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2021.

[54] J. Cao, B. Chen, C. Sun, L. Hu, and X. Peng, "Characterizing performance bugs in deep learning systems," *CoRR*, vol. abs/2112.01771, 2021. [Online]. Available: https://arxiv.org/abs/2112.01771