Challenges in Migrating Imperative Deep Learning Programs to Graph Execution: An Empirical Study

Tatiana Castro Vélez
City University of New York (CUNY) Graduate Center
USA
tcastrovelez@gradcenter.cuny.edu

Raffi Khatchadourian
City University of New York (CUNY) Hunter College
USA
raffi.khatchadourian@hunter.cuny.edu

Mehdi Bagherzadeh
Oakland University
USA
mbagherzadeh@oakland.edu

Anita Raja
City University of New York (CUNY) Hunter College
USA
anita.raja@hunter.cuny.edu

ABSTRACT

Efficiency is essential to support responsiveness w.r.t. ever-growing datasets, especially for Deep Learning (DL) systems. DL frameworks have traditionally embraced *deferred* execution-style DL code that supports symbolic, graph-based Deep Neural Network (DNN) computation. While scalable, such development tends to produce DL code that is error-prone, non-intuitive, and difficult to debug. Consequently, more natural, less error-prone imperative DL frameworks encouraging *eager* execution have emerged but at the expense of run-time performance. While hybrid approaches aim for the "best of both worlds," the challenges in applying them in the real world are largely unknown. We conduct a data-driven analysis of challenges—and resultant bugs—involved in writing reliable yet performant imperative DL code by studying 250 open-source projects, consisting of 19.7 MLOC, along with 470 and 446 manually examined code patches and bug reports, respectively. The results indicate that hybridization: (i) is prone to API misuse, (ii) can result in performance *degradation*—the opposite of its intention, and (iii) has limited application due to execution mode incompatibility. We put forth several recommendations, best practices, and anti-patterns for effectively hybridizing imperative DL code, potentially benefiting DL practitioners, API designers, tool developers, and educators.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Computing methodologies → Machine learning**.

KEYWORDS

empirical studies, deep learning, imperative programs, hybrid

ACM Reference Format:
Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. 2022. Challenges in Migrating Imperative Deep Learning Programs to Graph Execution: An Empirical Study. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Machine Learning (ML), including Deep Learning (DL), systems are pervasive in society. Central to such systems are dynamic models, whose behavior is ultimately defined by input data. However, as datasets grow, efficiency becomes essential to support responsiveness [101]. For industrial applications, DL frameworks—pillars of DL systems [49,51,63,99]—must quickly execute complex computations on large datasets while supporting easy-to-use programming paradigms [53]. For efficiency, DL frameworks have traditionally embraced a *deferred* execution-style that supports symbolic, graph-based Deep Neural Network (DNN) computation [21,38]. While scalable, development is error-prone, cumbersome, and produces programs that are difficult to debug [49,50,98,99]. Furthermore, because graph computation executes statements in a non-imperative order, traditional Software Engineering (SE) tools are inapplicable in troubleshooting bugs [8]. Contrarily, more natural, less error-prone, and easier-to-debug *imperative* DL programs [2,23,78] encouraging *eager* execution have emerged. Though ubiquitous, eagerly-executed imperative DL programs are slower, less efficient, and less scalable as their deferred-execution counterparts [21,29,35,53,68,78]. Executing (imperative) DL programs eagerly "makes tensor [matrix-like data structures central to DL] evaluation trivial but at the cost of lower performance" [66].[1] Thus, hybrid approaches [5,29,68]—integrated into mainstream DL frameworks—execute imperative DL programs as static graphs at run-time. For example, in *TensorFlow* [1]—a popular [46,99] DL framework—*AutoGraph* [68] can potentially enhance performance by decorating (annotating)—with optional yet influential decorator arguments—appropriate Python function(s) with `@tf.function`. Decorating functions with such hybridization Application Programming Interfaces (APIs) can increase imperative DL code performance without explicit modification.

Though promising, hybrid approaches necessitate non-trivial specialized metadata [53] and exhibit limitations and known issues [34] with native program constructs. Subtle considerations are required to make code amenable to safe, accurate, and efficient graph execution—avoiding performance bottlenecks and semantically inequivalent results. Therefore, developers are burdened

[1] Performance is this paper refers to *run-time* performance (speed), not model accuracy.

```
1 # Build a graph.
2 a = tf.constant(5.0)
3 b = tf.constant(6.0)
4 c = a * b
5 # Launch graph in a session.
6 sess = tf.Session()
7 # Evaluate the tensor `c`.
8 print(sess.run(c)) # prints 30.0
```

Listing 1: *TensorFlow* deferred execution-style code [40].

with making their code compatible with the underlying execution model conversion, as well *manually* specifying which functions that should be converted. While alternatives [53] exist, they impose custom Python interpreters, which may be impractical for industry, and support only specific Python constructs. Thus, there is a knowledge gap in how hybrid technologies are used in real-world DL applications, leading to the challenges in successfully applying them underexplored. Without this insight, DL systems may be inefficient, fallible, and difficult to maintain. Moreover, advances in DL are likely to be futile if they cannot be effectively used.

To fill this gap, we conduct an empirical study on common development challenges in migrating imperative DL code to graph execution using hybridization in open-source DL systems. Particularly, we set forth bug patterns and corresponding challenges involved in writing *reliable* yet *performant* imperative DL code. Such knowledge can help drive new automated migration techniques, IDE code completion, and automated (data science-specific [10,24,25]) refactoring mining approaches [92]. The results: (i) advance knowledge of this emerging yet pervasive hybrid paradigm, (ii) provide feedback to language and API designers for future API versions, (iii) help tool designers comprehend difficulties with writing performant imperative DL code, (iv) propose preliminary *recommendations*, *best practices*, and *anti-patterns* for practitioners in using hybridization effectively, and (v) assist educators in teaching hybridization APIs.

Our study involves analyzing occurrences of `tf.function` in 250 projects, consisting of 19.7 MLOC, along with 470 and 446 manually examined code patches (Git commits) and bug reports (GitHub issues), respectively. Challenges—along with their causes, symptoms, and fix patterns—are taxonomized using manual processes aided by automated software repository mining. Due to its popularity and its extensive analysis by previous work [22,48,49,51,63,74,98,99], we focus on hybridization in *TensorFlow*. Our study indicates that: (i) `tf.function` is widely used, (ii) misusing `tf.function` was a major theme in migrating imperative DL programs to graph execution, (iii) subtle bugs in using `tf.function` can result in performance *degradation*—the opposite of its intention, and (iv) `tf.function` is commonly incompatible in a given context—limiting its application.

Our contributions can be summarized as follows:

Hybridization bug hierarchical taxonomy From 470 and 446 patches and bug reports, respectively, of 250 projects manually examined, we build a rich hierarchical taxonomy of common hybridization usage challenges.

Recommendations, best practices, & anti-patterns We propose preliminary recommendations, best practices, and anti-patterns for effectively hybridizing imperative DL code from our statistical results, as well as an in-depth analysis.

Complete results of our study are available in our dataset [20].

2 MOTIVATING EXAMPLES & BACKGROUND

Popular DL frameworks have historically embraced *deferred* execution-style (low-level) APIs, making DNNs straight-forward to

```
1 class SequentialModel(tf.keras.Model):
2   def __init__(self, **kwargs):
3     super(SequentialModel, self).__init__(...)
4     self.flatten = layers.Flatten(
5       input_shape=(28, 28))
6     num_layers = 100 # Add many small layers.
7     self.layers = [layers.Dense(64, activation =
8       "relu") for n in range(num_layers)]
9     self.dropout = tf.keras.layers.Dropout(0.2)
10    self.dense_2 = tf.keras.layers.Dense(10)
11  @tf.function(...) # Executes
12    # the model as a graph
13    # (with optional args).
14  def __call__(self, x):
15    x = self.flatten(x)
16    for layer in self.layers:
17      x = layer(x)
18    x = self.dropout(x)
19    x = self.dense_2(x)
20    return x
```

Listing 2: *TensorFlow* imperative (OO) DL model code [35].

```
@tf.function
def f(x):
  print("Input: ", x)
  f(1)
  f(1)
  f(2)
```

Output (expecting 1, 1, 2):

```
Input: 1
Input: 2
```

Listing 3: Imperative *TensorFlow* code with Python side-effects [34].

execute as symbolic graphs that enable various run-time optimizations. For example, during graph building (lines 2–4 of lst. 1), line 4 does not execute until the `Session` created on line 6 is run on line 8. While efficient, legacy code using such APIs are cumbersome, error-prone, and difficult to debug, maintain [49,50,98,99]. Such APIs also do not natively support common imperative program constructs, e.g., iteration [4]. Contrarily, *eager* execution-style DL APIs [2,78] facilitating higher-level, imperative, and Object-Oriented (OO) [23] (Python) programs that are easier-to-debug, less error-prone, and more extensible have emerged. For instance, with eager execution, line 4 of lst. 1 would execute and immediately evaluate tensor `c`, foregoing the need of a session. In many DL frameworks, eager execution is now the default.

Despite the benefits, executing (imperative) DL programs eagerly comes at the cost of run-time performance [66]. Thus, hybridization approaches [5,29,68] that execute imperative DL programs as graphs at run-time have been integrated into mainstream DL frameworks. For example, lst. 2 portrays *TensorFlow* imperative (OO) DL code representing a modestly-sized model for classifying images. On line 11, *AutoGraph* [68] is used to potentially improve performance by decorating the model's `call()` method with `@tf.function`, possibly providing optional yet influential decorator arguments. At run-time, `call()`'s execution will be "traced" and an equivalent graph will be generated [34]. In this case, a speedup ($runtime_{old}/runtime_{new}$) of ~9.22, averaged over five runs, ensues [56].

As noted in §1, while promising, hybridization usage presents unique challenges [34,53] in ensuring that programs run reliably *and* efficiently. If used incorrectly, hybridization may yield programs that result in unexpected run-time behavior. Decorating the right functions, supplying the correct decorator arguments, using the appropriate API, and properly structuring imperative DL code so that it is amenable to graph execution can be daunting, especially for developers (data scientists) lacking SE expertise.

Python Side-effects. Problematic for `tf.function`-decorated functions[2] is side-effect producing, native Python statements, e.g., printing, list appending, global variable mutation [34]. Because they are traced, a function's behavior is "etched" into its corresponding graph and thus can have unexpectant results, executing side-effects multiple times or not at all. Side-effects occur when `tf.functions`

[2]Herein, "`tf.function`-decorated" functions will be referred to as "`tf.functions`."

```
1  class Model(tf.Module):
2    def __init__(self):
3      self.v = tf.Variable(0)
4      self.counter = 0
5
6    @tf.function
7    def __call__(self):
8      if self.counter == 0:
9        self.counter += 1
10       self.v.assign_add(1)
11     return self.v
12 m = Model()
13 for n in range(3):
14   print(m().numpy())
```

Output (expecting 1, 1, 1):

```
1
2
3
```

Listing 4: Imperative *TensorFlow* code using a counter [34].

```
model = SequentialModel()
res1 = model(tf.constant([1, 2, 3]))
res2 = model(tf.constant([1, 2, 3, 4,
  ↪ 5]))
```

```
WARNING: 5 out of the last 5 calls...
    triggered tf.function retracing.
    Tracing is expensive...
```

Listing 5: DL model (lst. 2) client code using varying datasets [34].

are called the first time; subsequent calls with similar arguments execute the graph instead. For example, on line 3 of lst. 3, `f()` outputs `x`. On line 1, `f()` is decorated with `@tf.function`, which migrates it to a graph at run-time. Then, `f()` is invoked three times, the first two with the argument `1` and the last with `2`. In the output on the right, the first invocation of `f()` on line 4 results in a graph being built (through tracing) that—due to a similar argument—is later used on line 5. Consequently, the side-effecting code on line 3 is *not* exercised. In contrast, line 3 *is* exercised as a result of the call on line 6 due to a different argument being supplied.

Although lst. 3 is simple, other times, unexpected behavior can be more difficult to notice. Consider lst. 4, where a model using a `counter` to safeguard a variable incrementation. The initial value of `counter`, however, is captured during tracing upon the first model invocation (line 14). The overall effect is that the value of `v` is incremented *unconditionally* (line 10) each time the model is invoked. Such problems are common among developers migrating deferred-execution–style DL code (e.g., lst. 1) to an imperative style (e.g., lst. 2). Worse yet, developers only realize such errors after observing suspicious numerical results or significantly lower performance than expected (e.g., when guarded operations are costly) [34].

When To Use Hybridization? Besides ensuring that DL code is amenable hybridization [16], developers must also know *when* and *where* to use it to avoid performance bottlenecks and other undesired behavior. For example, confusion exists on how often `@tf.function` should be applied [84], and calling `tf.functions` recursively could cause infinite loops [34]. Even if a recursion seems to work, the `tf.function` will be traced *multiple* times ("retracing") potentially impacting performance. Also, using `@tf.function` on small computations can be dominated by graph creation overhead [35].

Using Hybridization Parameters. Decorating the *correct* function but with *incorrect* decorator arguments may result in performance degradation. For instance, retracing helps ensure that the correct graphs are generated for each set of inputs; however, excessive retracing may cause code to run more slowly had `tf.function` *not* been used [34,81,96]. Lst. 5 depicts code that invokes the model declared in lst. 2 multiple times using different (hypothetical) datasets, which produces the warning on the right. To limit retracing, an `input_signature` can be specified on line 11, lst. 2 as follows:

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
```

A `[None]` dimension in the `tf.TensorSpec` allows for flexibility in trace (graph) reuse. Since tensors are matched on their shape, a `None` wild card allows `tf.functions` to reuse traces for variably-sized input—occurring when sequences or images are of different lengths or sizes, respectively. Since each call no longer produces a trace, the warning disappears—averting any performance bottlenecks.

These simplified examples demonstrate that effectively using hybridization is not always straight-forward, potentially requiring complex analyses and a thorough understanding of API intricacies—a compounding problem in more extensive programs. As imperative DL programming becomes more widespread, statistical insight into how such programs are best written efficiently and how to avoid common bugs would be extremely valuable to developers.

Table 1: Studied subjects.

	subj	KLOC	studied periods	cmts/iss	kws	exe
fixes	122	10,879	2015-11-06 to 2021-01-14	199,140	470	470
reports	167	17,378	2012-05-07 to 2021-08-11	237,232	704	446
Total	250*	19,677*	2012-05-07 to 2021-08-11	436,372	1,174	916

* Represents unique totals due to subject overlap between the study portions.

3 METHODOLOGY

Subjects. We examined both Git commit changesets (code patches; row **fixes**, tab. 1) representing bug fixes involving `tf.function` GitHub issues (row **reports**) mentioning `tf.function`. Our study encompassed 250 open-source DL systems (column **subj**), comprising ~19.7 million lines of source code (column **KLOC**), 199,140 Git commits (column **cmts** for **commits**), 237,232 GitHub issues (column **iss** for bug **reports**), and 460.21 years of combined project history, averaging 1.86 years per subject. Subject details may be found in our dataset [20]. Subjects vary in their domain and application, as well in their as size and popularity. Subjects sources are publicly available on GitHub, exhibit a variety of GitHub metrics, and include a mix of libraries, frameworks, and applications. While we focus `tf.function` client usages, we include *TensorFlow* as developers often file GitHub issues against it to discuss `tf.function` usage challenges and potential bugs. Subjects include those used in previous studies [22,24,48,49,51,52,63,98,99] and appearing in data science-specific datasets [17]. To determine if a project represents a DL system—with at least one DL module—like related work [52], we searched repositories for specific keywords, e.g., "keras," "layer," "net," "neural network," "deep learning." We then verified the code to ensure that the keywords represented DL contexts.

For changesets (bug fixes), subject criteria consists of having at least one commit whose changeset contains `tf.function`. For issues, subjects must have at least one GitHub issue mentioning "tf.function." Subjects were mostly written in Python, which is popular for DL [15]. While the subjects include popular open-source repositories from well-known and reputable organizations, e.g., Apache [6], Apple [7], Google [37], NVIDIA [75], they also include lesser-known repositories to understand hybridization challenges facing the DL community-at-large. Furthermore, hybridization is relatively new—`tf.function` was released on September 30, 2019.

Mining. To find changesets (patches) representing hybridization bug fixes, we mined repositories for commits referencing `tf.function` using `gitcproc` [19], a tool for classifying Git commits used by previous work [11,32,58,83,89,91]. Row **fixes**, column **kws** of tab. 1 is the commits containing `tf.function` in their changesets. We manually examined all 470 commits, portrayed by row **fixes**, column **exe**. To find issues related to hybridization, we mined repositories for GitHub issues mentioning "tf.function" by first filtering out issues containing only irrelevant discussion (e.g., "social conversation") using a classification model [9]. We then invoked the *GitHub Search API* [33] to select (open and closed) issues that included the keyword. To reduce false positives, since the API ignores punctuation, we further filtered the results to ensure that they included the period. Row **reports**, column **kws** of tab. 1 is the issues[3] containing "tf.function" in either their title or body (description and conversations). We randomly selected a subset of these to examine manually (details below), portrayed by row **reports**, column **exe**.

Identification. We used a feature of `gitcproc` that leverages heuristics based on commit log messages to identify commits that represent bug fixes. Natural language processing (NLP) is used to determine the commits that fall into this category. Doing so helps us to focus on likely bug fix commits for further manual examination. Random matching issues—with ones containing code being favored—were chosen for manual inspection to verify whether they involved hybridization challenges. Next, the authors manually examined the commits and issues to determine if they indeed relate to hybridization bugs and usage challenges. Two authors are SE and PL professors with extensive expertise in software evolution, system performance, and empirical SE. Another author is a data mining and ML professor with substantial proficiency in AI and SE. Three authors have several years of industrial SE experience.

Although the researchers did not converse during the initial identification and classification process to avoid bias, this mix of expertise is effective in studying SE tasks in DL systems. The researchers convened regularly during the study, as well as at the end for finalization, to solidify the results. As the authors did not always have detailed knowledge of the particular systems, only changes where a bug fix was extremely likely were marked as such. The authors also used commit comments and referenced bug databases to ascertain whether a change was a bug fix.

Classification. For commits, once bug fixes were identified, the authors studied the code changes to determine the category of bug fixes and whether the category relates to hybridization. For issues, the authors examined issue descriptions and discussions, paying attention to the `tf.function` challenges being described and their possible solutions and workarounds. Categories were then formed into a hierarchy, in part by using the *TensorFlow* documentation [34]. On several occasions, developers were contacted for clarification.

4 RESULTS

4.1 Quantitative Analysis

From the 470 commits and 446 GitHub issues (totaling 916) manually examined (column **exe**, tab. 1), we found 157 and 123 (totaling 280) `tf.function` client code bug fixes and developer challenges depicted in columns **cmts** (commits) and **iss** (GitHub issues) of tab. 2,

[3]Also includes pull (patch) requests as these are treated similarly in GitHub.

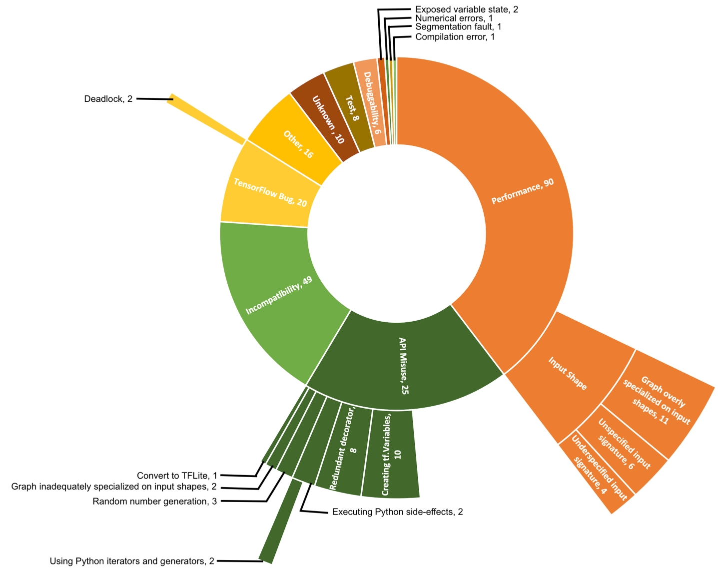

Figure 1: Discovered problem categories (hierarchical).

Table 2: Discovered top-level problem categories.

problem	abbr	cmts	iss	total
Performance	PRF	74	37	111
API misuse	APM	23	30	53
Incompatibility	INC	16	33	49
TensorFlow bug	TFB	4	18	22
Other	OTH	14	2	16
Unknown	UKN	10	0	10
Test	TST	8	0	8
Debuggability	DBG	4	2	6
Exposed variable state	EVS	1	1	2
Compilation error	CMP	1	0	1
Numerical errors	NME	1	0	1
Segmentation fault	SEG	1	0	1
Total		157	123	280

respectively. Finding these bugs and understanding their relevance required a significant amount of manual labor that may not be feasible in more large-scale, automated studies. Python, being a dynamic language, can be difficult to analyze, particularly w.r.t. inheritance relationships; subclassing Keras models is a common way to write imperative DL code in *TensorFlow* (cf. line 1, lst. 2). Furthermore, our number of findings (280) is comparable with previous studies involving manual inspection (e.g., Tang et al. [88] found 285, Zhang et al. [99] found 175, Khatchadourian et al. [58] found 61). Nevertheless, as `tf.function` becomes more popular, we expect its usage and number of related bugs to grow.

4.1.1 *Problem Categories.*

From the manual changes, we devise a set of common problem categories. Bug fixes and GitHub issues are grouped into these categories as shown in fig. 1 and tab. 2 (column **abbr** is the category abbreviation). The former includes combined data (commits and issues), while the latter separates the two.

Fig. 1 presents a hierarchical categorization—with varying levels of detail—of the 280 discovered `tf.function`-related challenges in our subjects. Challenges are represented by their problem category name and are followed by their counts. Categories without instances

Table 3: Performance fixes.

fix category	count
Add `tf.function` decorator	61
Change `tf.function` argument	20
Add `input_signature` argument to `tf.function`	9
Remove `tf.function` decorator	8
Upgrade to new library version	4
Relocate `tf.function` (use on different function)	5
Re-add `tf.function` decorator	2
Unsolved (open)	2
Total	111

are *abstract*, i.e., they *only* group together other categories. Tab. 2 portrays a nonhierarchical, top-level view of fig. 1.

Challenges are grouped into several (top-level) problem categories. Categories include performance (PRF, 90; further discussed later), API misuse (APM, 25; further discussed later), and incompatibility between execution modes, i.e., eager and deferred, where `tf.function` is used in a context not amenable to graph conversion (INC, 48; further discussed later). An example of the latter is where particular loss functions cannot be used in graph mode or there is an *AutoGraph* limitation that prevents graph conversion. Other problem categories include dealing with or working around open bugs related to `tf.function` in *TensorFlow* (TFB, 20; further discussed later) and "other" (OTH, 16), which involves syntactic corrections, general cleanup, and refactorings—a category similar to that used by previous work [58,91]. "Unknown" (UKN, 10) represents situations where the problem category was indeterminable without further domain knowledge or developer input. Only 3.57% of problems had unknown categories. Code changes involving `tf.function` appearing in tests were categorized as "Test" (TST, 8).

Debuggability. Debuggability (DBG, 6) represent situations where using `tf.function` to improve performance of DL code may, in turn, reduce a developer's ability to easily debug it. "In general, debugging code is easier in eager mode than inside `tf.function`" [34]. In such situations, developers may not understand that using `tf.function` is the reason why they are not able to debug their code, e.g., intermediate variable values may be missing. Or, `tf.function` may *temporarily* be removed (via a commit) to facilitate debugging, but developers inadvertently neglect to replace it (cf. §4.2.5). This latter situation is unfortunate as, to assist in the debugging process, a flag can be used to globally (temporarily) toggle `tf.function` [34].

Other Categories. Other (top-level) categories were more minor in terms of their counts, yet have potentially significant consequences. For example, exposed variable state (EVS, 2) occurs when saving (exposed) program state (variables) is problematic during `tf.function` conversion at run-time, e.g., variables becoming undefined [47]. Numerical errors (NME, 1) involve possible numeric overflow. Autograph compilation errors (CMP, 1) surface when `tf.functions` are compiled and subsequently result in compilation errors. This problem may arise when certain dynamic Python features, e.g., lexical scoping, are utilized (cf. §4.2.2). Segmentation fault (SEG, 1) is when using `tf.function` causes a program crash.

Performance. As the main purpose to hybridization is to improve the performance of imperative-style DL code by building a bridge to graph-based execution, it was not surprising that performance—at 39.64% (111/280)—was the largest category (finding 1, fig. 2). Performance problems represent a spectrum of situations, stemming from using `tf.function` to solve a DL code performance bug to not observing the expected speedup from using `tf.function` to exhibiting *worse* performance that *not* using `tf.function`. Tab. 3 portrays the various fixes used to solve performance problems. Though the majority of times it was used to improve performance of imperative DL code, we found that in 7.21% (8/111) of cases, `tf.function` was *removed* to alleviate performance problems, leading to finding 2. Moreover, only 54.95% of imperative DL code performance problems involving hybridization were fixed by *adding* `tf.function`. Thus, the remaining 45.05% of cases were due to *existing* usages of `tf.function`, leading to finding 3. In fact, 25.23% of performances fixes involved altering `tf.function` arguments, leading to finding 4.

Performance problems are further categorized into those related to "input shapes," which make up 18.92% of all performance problems (finding 5, fig. 2). Tensors are heavily used DL programs, and accurately matching tensor "shapes," i.e., dimensions, is often required to write reliable DL code. In hybridization, since `tf.functions` are being traced and thus converted into graphs, the underlying framework—by default—tries to build specialized graphs for particular kinds of input. However, when tensors are involved, graphs may be specialized to particular input shapes, creating a situation where function retracing is excessive. Retracing can lead to significant performance degradation [35] (cf. §2).

To curb this problem, an (`input_signature`) argument may be supplied to `tf.function` that specifies an *expected* range of shapes. In effect, developers provide contextual information to the framework about how `tf.functions` will be used. For instance, setting

(1) At 39.64% (111/280), performance problems was the largest category involving `tf.function` usage.
(2) Despite its intent to improve code performance, in 7.21% (8/111) of cases, `tf.function` *caused* performance degradation.
(3) Only 54.95% (61/111) of imperative DL code performance problems were fixed by *adding* `@tf.function`. The remaining 45.05% were due to *existing* usages of `tf.function`.
(4) 25.23% of performance fixes involve altering developer-supplied arguments to `tf.function(...)`.
(5) 18.92% of performance problems involve incorrect input tensor shape specifications.
(6) At 18.93%, API misuse—using `tf.function` inconsistently w.r.t. its documentation—was the second largest problem category.
(7) 37.74% (20/53) of API misuse was caused by developers not understanding how to use hybridization APIs correctly.
(8) To fix API misuse, `tf.function` was *removed* in 28.30% of cases. In 46.67% of these cases, developers abandoned hybridization due to API confusion, of which 62.50% resulted in run-time errors.
(9) Execution mode incompatibility—at 17.50% (49/280)—was the third largest problem category, meaning that developers struggle to seamlessly use similar constructs between different modes.
(10) 81.63% of incompatibility problems lead to run-time errors or unexpected results, which do not surface until *after* running the code.
(11) Bugs with *TensorFlow*—for which developers were offered a workaround or waited for a new framework version—made up 7.86% of discovered problems. Of these, 9.09% involve deadlocks.

Figure 2: Study findings.

Table 4: API misuse causes.

cause	count
API confusion	20
Use of graph mode	14
Decorated outer function calls unnecessarily decorated inner function	8
Incorrect `tf.function` argument	7
Use of eager mode	2
Lost variable state due to graph conversion	1
Lack of static shape specifications	1
Total	53

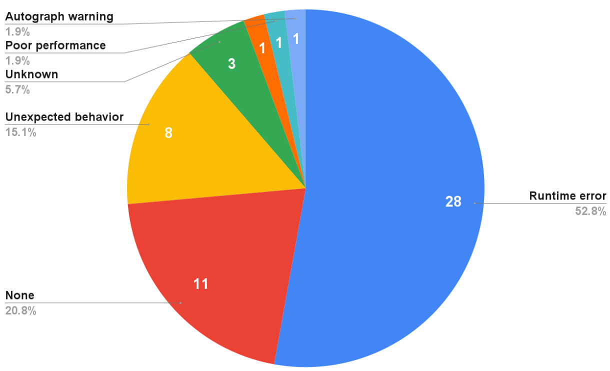

Figure 3: API misuse symptoms.

`experimental_relax_shapes` to **`True`** may cause `tf.functions` to generate fewer graphs that are less specialized on input shapes. However, this may not match reality, especially when dealing with dynamic shapes. As such, we further divide "input shape" challenges:

Graph *overly* specified on input shapes (11) Generated graphs are too specific for the context where a `tf.function` is being used, which can occur when either:
(i) `experimental_relax_shapes` is incorrectly set to **`False`**.
(ii) `input_signature` is unnecessarily specified. Either it should be either removed or set to **`None`** (the default).

***Under*specified input signature (4)** The `input_signature` parameter lacks proper arguments to avoid excessive retracing.

***Un*specified input signature (6)** The `input_signature` is missing in contexts that are advantageous to graph specialization.

API Misuse. API Misuse—the second largest problem category at 18.93%—involves situations where `tf.function` is not used in a way recommended by the API documentation (finding 6, fig. 2). Misusing APIs typically results in either run-time errors or unexpected behavior. Violating DL API constraints may lead to crashes and poor performance [49,51]. In high-level, e.g., imperative DL, code, bugs are commonly due to misunderstandings of the guarantees offered and obligations imposed by increasingly layered software, e.g., those written against the *TensorFlow* API [61]. *TensorFlow* documentation contains a prominent sections regarding `tf.function` and *AutoGraph* usage constraints and limitations. If such constraints, e.g., w.r.t. control-flow, side-effects, global variables, are violated, *AutoGraph* will not properly generate graphs from Python code. Despite the vast documentation, at 37.74%, API confusion was the largest cause of API misuse (tab. 4), leading to finding 7.

We found that the most common way (28.30% or 15/53) to fix API misuse was to *remove* `@tf.function`. Of these, in 46.67% of cases (7/15), the problem cause was that `@tf.function` was used to decorate an *inner* function called by an *already* decorated *outer* function. As `tf.function` applies to the decorated function *and* all other functions it calls and since the inner function cannot be called from any other function besides the outer function, the inner function decorator is unnecessary and can thus be safely removed [35]. However, another 46.67% (7/15) of cases were caused by API confusion. Thus, in these cases, unfortunately, developers *abandoned* `@tf.function`—along with its potential to enhance performance—due to their confusion over how to use it. Most likely, developers were doing so to avoid run-time errors, which occurred in 62.50% (5/8) of `tf.function` removals not caused by unnecessary inner function decoration and 52.83% (28/53) overall (fig. 3), leading to finding 8.

API misuse is further divided into several categories, the largest of which involves creating `tf.Variables` within `tf.functions` (10). A `tf.Variable` represents a tensor whose value is mutable [36]. Currently, `tf.function` only supports singleton `tf.Variables`; creating multiple `tf.Variables` within the scope of a `tf.function` results in a run-time exception [34]. Redundant decoration (8) is where *multiple* functions on a call path are unnecessary decorated with `@tf.function`; all functions called from a `tf.function` are also automatically migrated to graphs. Accurately approximating such paths statically—especially in the context of a dynamic language such as Python—may be difficult, and ample confusion among developers on where to apply `@tf.function` exists [84] (cf. §2).

Executing Python side-effects (2) refers to the situation where `tf.functions` contain side-effect producing Python statements. As described in §2, executing such statements within migrated graphs can have unexpectant results, sometimes executing twice or not all. A specific pattern of side-effects were those involving the use of iterators and generators (2), a common looping mechanism in Python code. Random number generation (RNG, 3) problems occur when developers do not use RNG facilities consistently with the documentation, commonly resulting in unexpected behavior under graph mode. For example, RNG creation inside a `tf.function` can only happen during the first run of the function [39]. Seeding may also not work as expected in graph mode (e.g., [31])—"when [a] global seed is set but [*TensorFlow*] operation seeds are not, the sequence of random numbers are the same for each `tf.function`" [43]. "Graph *inadequately* specialized on input shapes" (2) involves an API misuse that is opposite to the "graph *overly* specified on input shapes" *performance* problem category described earlier. Such problems may be fixed by setting `experimental_relax_shapes` to **`False`** (the default). In other words, the shape specification is too *general*, which may result in a situation that is not amenable to graph migration [79]. For example, an `input_signature` may be supplied using a wild card shape to improve performance (q.v. §2) but results in a run-time error due to a tensor dimension mismatch [34,97]. Conversion to TFLite (1) represents problems with an alternate use case of `tf.function` to convert a DL model to a portable format.

Execution Mode Incompatibility. At 17.50%, incompatibility is the third largest problem category, leading to finding 9, fig. 2. Developers seemingly struggle with seamlessly using imperative DL program constructs, e.g., particular loss functions, across execution modes. Ideally, developers would be able to toggle between eager

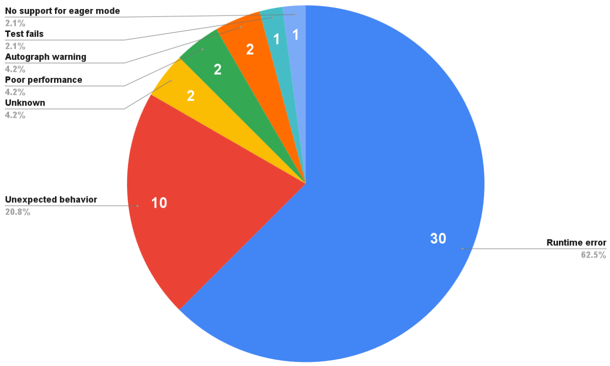

Figure 4: Incompatibility problem symptoms.

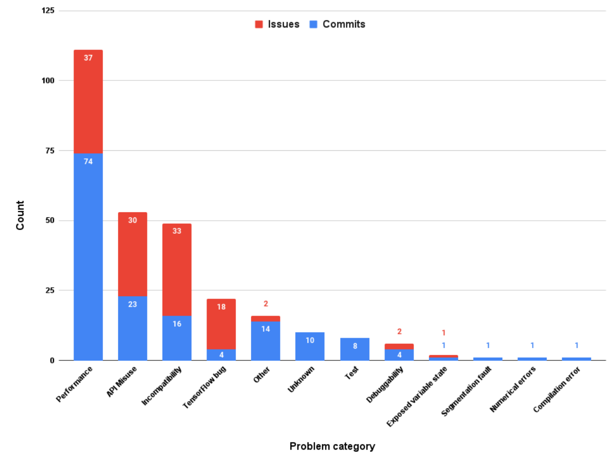

Figure 5: Top-level problem category comparison.

and graph execution modes, with *AutoGraph* simply enhancing performance without necessitating client code modifications. In other words, incompatibility problems prevents developers from focusing on the correctness of their DL code—thinking of performance as an afterthought. Instead, in using hybridization, developers need to be cognizant of how their DL code is being migrated to graphs and the *internals* of hybridization. Moreover, developers must (manually) be aware of which constructs are amenable to graph conversion, how best to write code that works in either mode, and how to interact with code that may be executed in a different mode.

Execution mode incompatibility problems have dire consequences. As shown in fig. 4, 81.63% of symptoms resulting from incompatibility involve run-time errors or unexpected behavior. Such problems that only occur at run-time are difficult to uncover and, if found, may be found after deployment, leading to finding 10, fig. 2.

TensorFlow Bugs. *TensorFlow* bugs (TFB) made up 7.86% of discovered bugs, leading to finding 11, fig. 2. Such bugs involve dealing with or working around open *TensorFlow* bugs related to `tf.function`. As hybridization is relatively new, the `tf.function` API is under active development. Thus, it was not uncommon for developers to report bugs with `tf.function` to *TensorFlow* by filing issues against the *TensorFlow* GitHub repository; 81.82% of TFBs appear as GitHub issues (see tab. 2 and fig. 5). We categorize bugs as TFB if they were in fact real bugs with *TensorFlow* that required a workaround—often suggested by *TensorFlow* contributors—or a new *TensorFlow* library version to solve. If the reported bugs were not resolved to be the result of problems with *TensorFlow*, such bugs were not categorized as TFB but perhaps other categories.

TFB is further categorized into deadlock (2). Situations leading to the execution of a `tf.function` being deadlocked include using tensors as stopping condition of a recursive `tf.function` [94]. Deadlock may also occur as a result of other, specific `tf.function` client code patterns causing the *TensorFlow* run time to deadlock, e.g., calling a `tf.function` from a `tf.py_function` [95], which executes native Python functions as graph operations eagerly [42].

4.1.2 Commits vs. GitHub Issues. Fig. 5 compares the different sources—commits (bottom/blue bars) and GitHub issues (top/red bars)—of problem categories. Performance—the largest problem area—was 2/3 more likely to appear in *commits* vs. issues. In contrast, "incompatibility" was 2/3 more likely to appear in *issues* vs. commits. Notably, 80% of TFB problems were found in GitHub issues compared to commits. Lastly, all UKN and TST bugs were found in commits. §5 discusses possible reasons for these differences.

(1) Favor `@tf.function` on Python functions containing imperative, otherwise eagerly-executed, DL code to improve performance.
(2) If possible, supply an `input_signature` argument to `tf.function` with the intended shape and types of any input tensors to avert retracing—a practice similar to that of providing type annotations to variables in dynamic languages to assist with type inferencing.
(3) When an operation is deemed incompatible with hybridization, check the documentation to see if additional steps are required to make the imperative DL code more amenable to graph conversion.
(4) Framework limitations may impede performance enhancements. Check for potential workarounds of (unresolved) *TensorFlow* bugs.
(5) Use `tf.config.run_functions_eagerly(True)` to *temporarily* disable `tf.function` to facilitate debugging.

Figure 6: Preliminary hybridization best practices.

(1) Hybridizing nested functions may cause performance degradation. Sacrifice some modularity by either hybridizing the top-level function or refactoring the nested function to a top-level function [86].
(2) Using `tf.Variables` inside `tf.functions`, directly or indirectly.
(3) Leveraging dynamic language features, e.g., lexical scoping, inside `tf.functions`, directly or indirectly, where they are unavailable.

Figure 7: Preliminary hybridization anti-patterns.

```
1 + @tf.function
2   def pm(linear):
3     state = lpt_init(linear, a0=0.1, order=1)
4     final_state = nbody(state,  stages, nc)
5     tfinal_field = cic_paint(tf.zeros_like(linear), final_state[0])
6     return tfinal_field
```

Listing 6: Commit `af1664e7`in `galference`: bug boxsize=nc

4.2 Qualitative Analysis

4.2.1 Performance. In §4.2.1, `pm()` is decorated with `@tf.function` (line 1). Using `tf.function` "ensure[s] that the graph for [a] function

```
1 - @tf.function
2 + @tf.function(input_signature=[
3 +   tf.TensorSpec(shape=(None, self.num_states), dtype=tf.float32),
4 +   tf.TensorSpec(shape=(None, self.num_actions), dtype=tf.float32),
5 +   tf.TensorSpec(shape=(None, 1), dtype=tf.float32),
6 +   tf.TensorSpec(shape=(None, self.num_states), dtype=tf.float32),])
7   def update_weights(s, a, r, sn): # ...
```

Listing 7: Commit `02a3f297`in `DDPG-tf2`: Fixed all … this should work

```
1    def ndiagquad(funcs, H: int, Fmu, Fvar, logspace: bool = False, **Ys):
2      # Computes N Gaussian expectation integrals of one or more functions ...
3  -   def unify(f_list): # Stack a list of means/vars into a full block.
4  -     return tf.reshape(tf.concat([tf.reshape(f, (-1, 1)) for f in f_list],
5  -       axis=1), (-1, 1, Din))
6      if isinstance(Fmu, (tuple, list)):
7        Din = len(Fmu)
8  +     def unify(f_list): # Stack a list of means/vars into a full block.
9  +       return tf.reshape(tf.concat([tf.reshape(f, (-1, 1)) for f in f_list],
10 +         axis=1), (-1, 1, Din))
11       Fmu, Fvar = map(unify, [Fmu, Fvar])  # both [N, 1, Din]
```

Listing 8: Commit `b65848a2`in `GPflow`: fix compilation issue…

is compiled once and not every time it is called, thus gaining in speed and performance" [67], leading to best practice 1, fig. 6.

While hybridization can enhance the performance of their imperative, otherwise eagerly-executed, DL code, we found that developers struggled to use it correctly. Some distrusted it, stating, e.g., that "it does far too much hidden magic" [80]. Others [69] struggled with uncontrolled retracing (q.v. § 2 and 4.1.1), which actually results in worse performance—speedup of 0.13 in this case—by using `tf.function` than not using it: "`tfa.image.equalize()` uses an internal `scale_channel()` function[,] which triggers excessive retracing …." The problem is related to hybridizing *inner* functions: "I … tried using `@tf.function` at the `scale_channel()` and … `equalize_image()` level[s], but the further I moved it 'inside,' the slower `equalize()` became" [70]. The fix involved "using `@tf.function` at the top-level of `equalize()`, which made it run ~25%–40% faster …." The root cause is that, "using [embedded] functions ([i.e.,] defining functions inside function) will retrace the graph multiple times [as] the[ir] scope is not [publicly] visible, and the graphs cannot be cached" [86]. As a modularity mechanism, embedding (nesting) function definitions is a common idiom in Python, yet, currently, *TensorFlow* documentation does not mention this problem. Developers are left to consider the internals of *AutoGraph* in writing performant imperative DL code, leading to anti-pattern 1, fig. 7.

Input Signatures. Arguments to `tf.function()`, particularly involving input tensor shapes, may also influence performance (q.v. §2). §4.2.1 portrays an underspecified input signature (q.v. §4.1.1)—one of the most used `tf.function` parameters that we observed. On lines 2–6, a performance regression was fixed by adding an `input_signature` to a weight distribution `tf.function` to "make sure it does not recreate graph, which will slow down training significantly" [60]. The sequence of `tf.TensorSpecs` specifies the intended tensor shapes and data types (`dtypes`) that will be supplied to `update_weights()`. Otherwise, a separate (concrete) function (graph) is instantiated for each *inferred* input signature, which—depending on context—may result in retracing, leading to best practice 2, fig. 6.

4.2.2 Compilation Errors. Consider `unify()`—originally defined on lines 3–5, §4.2.2—that accesses `Din` on line 5. This variable, however, it is defined *after* the function definition on line 7, which is legal due to Python's lexical scoping rules. In other words, the value of `Din` will come from the *calling* context. In this case, `Din` on line 5

```
1 - @tf.function
2   def interpolate_bilinear(grid, query_points, indexing="ij", name=None): # ...
3     tf.debugging.assert_equal(query_shape[2], 2, message=
4       "Query points must be size 2 in dim 2.")
```

Listing 9: Commit `8bab3226`in `tensorflow/addons`: remove tf.func

```
1 - @tf.function
2   def train_step(image):
3     with tf.GradientTape() as tape:
4       outputs = extractor(image)
5       loss = style_content_loss(outputs)
6       loss += total_variation_weight * tf.image.total_variation(image)
7     grad = tape.gradient(loss, image)
8     opt.apply_gradients([(grad, image)])
9     image.assign(clip_0_1(image))
```

Listing 10: Commit `8bab3226`in `neuro-art`: Multiple request bugfix…

is replaced with the value defined on line 7 due to `unify()` being accessed on line 11. The code, though, results in the following (run-time) **`NameError`** on line 5: `free variable 'Din' referenced before assignment in enclosing scope` [26]. The problem is that, while it itself is not a `tf.function`, `ndiagquad()` is *called* by a `tf.function` elsewhere—it will also be compiled into a (static) graph (cf. §4.1.1). Thus, traditional dynamic language features such as lexical scoping are not available in static contexts. As a result, `unify()` is moved to line 8, where `Din` is in its declaration scope. Although Python is a dynamic language, developers must be aware that certain code will be compiled to static graphs, leading to anti-pattern 3, fig. 7.

4.2.3 API Misuse. On line 1 in §4.2.3, `@tf.function` is removed to fix a bug that is causing flaky tests [71]. The problem is deemed to be that `@tf.function` and the assert statement on lines 3–4 is incompatible. The developers express that "removing the decorator is not ideal, but stability is more important than the [speedup] we [would] get with [it]" [72]. However, this code likely causes a race condition because of a missing control dependency following the assertion. To use the assertion within a `tf.function`, a control dependency is required "to block follow-up computation[s] until the check has executed" as a result of the function being converted to a (static) graph [41]. This leads to best practice 3, fig. 6.

4.2.4 TensorFlow Bugs. On line 1, §4.2.4, `@tf.function` is once again removed. The problem is that—with `@tf.function`—the application "can only process one image before" needing to restarted [59], terminating with the message: `ValueError: tf.function-decorated function tried to create variables on non-first call`. Recall from §4.1.1 that shared variables inside a `tf.function` must be singleton; a run-time exception ensues otherwise [34]. However, it is not obvious from §4.2.4 where the variable creation occurs—there are no explicit `tf.Variables`. The developer expresses that "removing the … decorator is a viable workaround but not [a] best practice," and that the root cause is an (unresolved) *TensorFlow* bug [85]. In terms of §4.2.4, the problematic line is 8, as "calling `apply_gradients()` on an optimizer for the first time will create its internal variables" [77]. In terms of the framework, it transpires to be related to software layering, as, "sadly[,] there [is] currently no public API to just initialize the optimizer state but not [apply it]." While several developers found workarounds for their particular situations, imperative DL code such as that in §4.2.4 have foregone any potential performance gains resulting from `@tf.function`, leading to best practice 4 and anti-pattern 2 in fig. 6 and 7, respectively.

```
1 - @tf.function
2   def call(self, inputs):
3     """Call `Layer`"""
4 -   if not self.initialized:
5 -     self._data_dep_init(inputs)
6 +   if not self._initialized:
7 +     self._initialize_weights(inputs)
8     self._compute_weights() # Recompute weights for each forward pass ...
```

Listing 11: Commit `16ee6c59`in `tensorflow/addons`: tf.func for debug

(1) More tool-support for assisting with using `tf.function` may help produce reliable yet performant imperative DL code.
(2) Modernize and reformulate existing tensor shape mismatch detectors for imperative DL code and `tf.function(...)` input shapes.
(3) More formal specification in a design-by-contract (DbC) style may be helpful for new tool-support aimed to alleviate API misuse.
(4) Testing (dynamic analysis) focused on hybridized (imperative) DL code that runs under *multiple* execution modes may localize bugs.

Figure 8: Preliminary hybridization recommendations.

4.2.5 Debuggability. To improve debuggability, `@tf.function` is removed on line 1, §4.2.5 (cf. §4.1.1). However, in the latest file version, `@tf.function` has not been replaced. Thus, the developer may have inadvertently sacrificed permanent performance gains for temporary debuggability, leading to best practice 5, fig. 6.

5 DISCUSSION

Performance. It is not surprising that performance is our largest category (finding 1, fig. 2) since hybridization is centrally related to performance enhancement. The volume of performance problems is a testament to the struggles developers have in writing performant, imperative DL code. However, 45.05% of performance problems (finding 3) were due to *existing* `tf.function` usages, suggesting that developers also struggle with using hybridization *effectively* to achieve the performance they desire. A feasible explanation is that developers must *manually* decide: (i) where and when to use `tf.function`, (ii) the arguments to supply `tf.function` for their code to perform optimally, and (iii) which code is amenable to (efficient) graph conversion and which is not, all of which can be error-prone.

The overarching goal is reliable *and* performant DL code; reliability stems from being able to write DL code in a less error-prone imperative-style, while performance is achieved in migrating that code to graph execution at run-time. *AutoGraph*, as well as other hybridization technologies, attempt to achieve this goal by automating the migration process as much as possible—frequently requiring contextual information from developers as to their intentions and imposing limitations of where the technology can be used. The end result is a trade-off—one of many typically made by DL frameworks [50]. As discussed in §1, others [53] attempt to automate the entire migration process—not requiring any contextual metadata—but impose new trade-offs, such as necessitating custom Python interpreters that may not be practical for industrial applications and support only specific Python constructs.

As *AutoGraph* and other hybridization technologies are pervasively used, as well as being integrated into official distributions of popular DL frameworks, our suggestion is to retain (and continually improve upon) hybridization platforms, while simultaneously posing this problem as one of (API) *usability*. Our perspective is that what is needed it tool-support that will guide developers in using this technology correctly given a particular context, as well as automated refactoring and other source code transformation tools that can detect and repair hybridization problems. Such techniques would alleviate hybridization issues well-before they are seen beyond (production) deployment or after long training sessions, leading to recommendation 1, fig. 8.

Cao et al. [18] and Zhang et al. [99] also study performance of DL code and found that a modest portion of *non-imperative TensorFlow* program bugs involved performance problems. However, these problems were caused by confusion with the underlying computation model, which essentially requires developers to build graphs *manually*. In our case, graphs are built *automatically*. Since imperative DL code runs eagerly by default, it is understandable that our study would uncover more performance problems. In fact, Tambon et al. [87] also observe performance degradation of imperative DL code. Wan et al. [93] found that performance bugs took the longest average time to fix in blockchain systems.

Per finding 4, developer-supplied arguments to `tf.function()` played a major role in performance problems, comprising 25.23% of performance fixes. Furthermore, per finding 5, a significant percentage (18.92%) of performance problems involved parameters representing input shape specification—one of the most frequently used `tf.function` parameters (q.v. §4.2.1). Input shape problems are a central focus of related work [49,61,99] on DL programs; related studies [48,49,51,99] also found shape problems. A feasible explanation is that developers are challenged to determine tensor shapes from all possible call sites statically. We again advocate for more tool-support in this area, e.g., an adaptation of Lagouvardos et al. [61] for imperative DL programs focused on hybridization parameters, leading to recommendation 2 in fig. 8.

API Misuse. Per finding 6, using the `tf.function` API inconsistently with its documentation was a major theme. Feasible explanations include: (i) DL APIs—along with their documentation [45]—are particularly vast and complex [50], (ii) often, documentation consumers (developers) are not software experts [45], (iii) although developers are writing imperative DL code, there exist situations where they must nevertheless be cognizant of hybridization limitations, and (iv) error messages may not be helpful. Due to Item (i), learning how to use DL APIs effectively necessitates a steep learning curve, especially considering that hybridization is relatively new. As ML systems have a quick time-to-market [82], developers may be not have the luxury of time to thoroughly understand the documentation. This is especially evident in finding 7, with 37.74% of misuses caused by API confusion. Item (ii) has been recognized by other ML/DL software studies (e.g., [88]). We conjecture that Item (iii) can also be alleviated with more tool-support, however, such tool-support in this context may require (e.g., design-by-contract) formalization of DL API specifications (e.g., modeling operation limitations in particular contexts), leading to recommendation 3, fig. 8. A potential downside to recommendation 3 is the rapid change of ML APIs [24]. For Item (iv), developers often expressed frustration with error messages, e.g., "the main complexity in [*TensorFlow*] 2 is in `@tf.function`[;] … error messages should be as clear as possible, especially for common problems" [30].

Zhang et al. [98] likewise observed broader API misuse in DL systems. Nadi et al. [73] also found API misuse despite ample documentation in the context of cryptography—developers prefer higher-level documentation. Current hybridization documentation tends to focus on lower-level details—future research may explore whether a similar concept will work for DL APIs. Furthermore, our findings coincide with Jin et al. [54] that many performance bugs are due performance implication misunderstandings of certain functions.

Incompatibility. Execution incompatibility of particular Python constructs was also a major theme (finding 9, q.v. §4.1.1). Zhang et al. [98] found a similar problem in DL systems w.r.t. CPU/GPU compatibility. We again advocate for more automation to circumvent such problems. To use hybridization effectively, developers must understand which constructs are amenable to *both* eager and graph execution and make appropriate considerations. Tool-support, e.g., IDE recommendations, may be helpful here. To alleviate run-time errors and unexpected results, we also advocate for more testing (dynamic analysis) of (imperative) DL code that runs the same code under *multiple* execution modes. Testing of DL systems is an emerging yet promising area, and testing focusing on (imperative) DL code hybridization may help to shed light on: (i) where developers struggle to write performant yet reliable (imperative) DL code and (ii) potential areas of where hybridization technologies can be improved. This leads to recommendation 4, fig. 8.

Commits vs. GitHub Issues. That performance bugs appearing more in commits than GitHub issues may be due to the main way of improving performance—a code change, which can be subsequently benchmarked. Contrarily, "incompatibility" is more difficult to quantify, often resulting in unexpected behavior or run-time errors (q.v. fig. 4). Therefore, developers may be more likely to seek external assistance. Developers commonly file GitHub issues against *TensorFlow*; 93.75% of TFB issues are against the `TensorFlow` subject. That all UKN and TST bugs appeared in commits may be due to GitHub issues being easier to categorize than changesets and DL testing remains an emerging area, respectively.

6 THREATS TO VALIDITY

Subjects may not be representative of DL systems. To mitigate this, subjects encompass diverse domains and sizes, have been used in previous studies, and are from a data science-specific dataset (q.v. §3). Various GitHub metrics and DL-related keywords were used in choosing subjects. Also, hybridization is relatively new; we expect a larger selection of subjects as it grows in popularity.

Our study involved many hours of manual validation to understand and categorize bugs. To mitigate bias, we investigated referenced resources and comments made by developers to help more fully understand the challenges faced. The NLP of `gitcproc` may have missed bug fix changesets. Nevertheless, using it, we were still able to find 157 bugs (280 overall) that contributed to a rich bug categorization, best practices, and anti-patterns. Furthermore, `gitcproc` has been used previously in other studies (q.v. §3).

Hybridization in comparable DL frameworks may have yielded different challenges. Nevertheless, focusing on *TensorFlow* enables us to more thoroughly understand the intricacies involved in using hybridization effectively. Moreover, *TensorFlow* is a widely-studied and popular (industrial) DL framework (q.v. §1).

7 RELATED WORK

Cao et al. [18] characterizing performance bugs in DL systems. During their analysis of general performance bugs, they also find that developers often struggle with knowing where to add `@tf.function` and how to implement decorated functions for optimal performance. Beyond performance bugs, our study includes a rich, hierarchical taxonomy of varying hybridization bug types, including input shape mismatches, API misuse, and construct incompatibility, whose results include run-time errors, unexpected behavior, and deadlock. Tambon et al. [87] examine (silent) behavioral bugs *within* DL frameworks and their impact on client code. Their work is reminiscent of our TFB problem category (q.v. §4.1.1) and also note that performance degradation may lead to significant problems at run-time. While they do not explicitly mention hybridization performance bugs, some of their performance bugs in imperative DL code may be alleviate by using `@tf.function`. Zhang et al. [100] study API change trends in *TensorFlow* and for which reasons; our focus is on *client* code modifications involving hybridization. Baker et al. [14] extract 11 common *TensorFlow* API misuse patterns. Only one of the patterns (and corresponding fix suggestion) involves (a specific use case of) `tf.function`. In contrast, our study goes beyond API misuse and entails 12 top-level problem categories—24 overall—encompassing hybridization usage challenges.

Zhang et al. [98] present a large-scale empirical study of general DL questions on Stack Overflow. Particularly, their "CPU/GPU incompatibility" problem category resembles our execution mode incompatibility category. Concerning hybridization, whether the migrated graph executes on a GPU is typically decided by the underlying DL framework; our focus is on conversion itself. Islam et al. [49] and Zhang et al. [99] study general DL bug characteristics and present anti-patterns to avoid bugs. Islam et al. [51] study patterns in which such bugs are fixed. Chen et al. [22] explore faults in deploying DL models to mobile applications. Nikanjam and Khomh [74] catalog various design smells in DL systems and recommend suitable refactorings. Jebnoun et al. [52] correlate code smells with bugs in DL code. Liu et al. [63] characterize technical debt in DL frameworks, while Humbatova et al. [48] taxonomize (functional) faults in DL systems. Arpteg et al. [8] categorize (general) SE challenges in DL systems into three areas—development, production, and organizational. Liu et al. [62] study failed *TensorFlow* industrial jobs and propose a constraint-based approach for detecting shape-related errors. Amershi et al. [3] conduct a study at Microsoft, observing software teams as they developed AI applications. Lwakatare et al. [64] also classify SE challenges for ML systems at six different companies, focusing mainly on deployment issues. Thung et al. [90] examine bugs in three general ML systems, finding that nonfunctional bugs, of which performance problems may be categorized, require the most involved fixes. Dilhara et al. [24] study ML library evolution and its resulting client-code modifications. And, Dilhara et al. [25] and Tang et al. [88] analyze repetitive code changes and refactorings made in ML systems, respectively. While valuable, these studies do not deal with challenges faced in migrating imperative DL code to graph execution.

Several studies involve performance in other contexts. Han and Yu [44] study configurability and performance. Future work entails correlating their findings with `tf.function` arguments. Jin et al. [55]

study performance slowdowns caused by system side inefficiencies. Bagherzadeh et al. [12] investigate performance in Actor-based systems. Others study language features. Parnin et al. [76] study Java generics adoption. Dyer et al. [27] study language feature usage evolution. Khatchadourian and Masuhara [57] empirically assess default methods. There are many general empirical studies. Makhshari and Mesbah [65] taxonomize development challenges of IoT systems. Bagherzadeh and Khatchadourian [13] investigate common questions asked by big data developers, and Khatchadourian et al. [58] examine the use and misuse of Java streams. Engler et al. [28] and Tian and Ray [91] study errors in systems code.

8 CONCLUSION & FUTURE WORK

This study advances knowledge of the development challenges involved in migrating imperative DL code to graph execution via hybridization. A hierarchical taxonomy of common hybridization challenges was formulated and preliminary recommendations, best practices, and anti-patterns were proposed. In the future, we will explore additionally analyzing alternative developer resources, e.g., Stack Overflow, and integrating our results into automated bug finders and refactoring detection approaches [10,92].

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: a system for large-scale Machine Learning. In *Symposium on Operating Systems Design and Implementation*.

[2] Akshay Agrawal et al. 2019. TensorFlow Eager: a multi-stage, Python-embedded DSL for Machine Learning. (2019). arXiv: 1903.01855 [cs.PL].

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: a case study. In *International Conference on Software Engineering. Software Engineering in Practice*. IEEE. IEEE, (May 2019), 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042.

[4] Apache. 2018. Customer layers (beginners). Apache MXNet documentation. Retrieved 07/23/2021 from https://mxnet.apache.org/versions/1.7/api/python/docs/tutorials/packages/gluon/blocks/custom_layer_beginners.html.

[5] Apache. 2021. Hybridize. Apache MXNet documentation. (April 8, 2021). Retrieved 04/08/2021 from https://mxnet.apache.org/versions/1.8.0/api/python/docs/tutorials/packages/gluon/blocks/hybridize.html.

[6] Apache Software Foundation. 2021. Open Deep Learning compiler stack. (December 1, 2021). Retrieved 12/01/2021 from https://git.io/JMric.

[7] Apple Inc. 2021. Core ML tools. (December 1, 2021). Retrieved 12/01/2021 from https://git.io/JMr61.

[8] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch. 2018. Software Engineering challenges of Deep Learning. In *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 50–59. DOI: 10.1109/SEAA.2018.00018.

[9] Deeksha Arya, Wenting Wang, Jin L. C. Guo, and Jinghui Cheng. 2019. Analysis and detection of information types of open source software issue discussions. In *International Conference on Software Engineering*. (May 2019), 454–464. DOI: 10.1109/ICSE.2019.00058.

[10] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PyRef: refactoring detection in Python projects. In *International Working Conference on Source Code Analysis and Manipulation*. DOI: 10.1109/SCAM52516.2021.00025.

[11] Hlib Babii, Julian Aron Prenner, Laurin Stricker, Anjan Karmakar, Andrea Janes, and Romain Robbes. 2021. Mining software repositories with a collaborative heuristic repository. In *International Conference on Software Engineering* (ICSE-NIER '21). IEEE, 106–110. DOI: 10.1109/ICSE-NIER52604.2021.00030.

[12] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 214, (November 2020), 1–32. DOI: 10.1145/3428282.

[13] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going big: a large-scale study on what big data developers ask. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 432–442. DOI: 10.1145/3338906.3338939.

[14] Wilson Baker, Michael O'Connor, Seyed Reza Shahamiri, and Valerio Terragni. 2022. Detect, fix, and verify TensorFlow API misuses. In *International Conference on Software Analysis, Evolution and Reengineering*, 1–5.

[15] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The open-closed principle of modern Machine Learning frameworks. In *Mining Software Repositories*, 353–363. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196445.

[16] Joel Berkeley. 2020. Ensure compatibility with tf.function. secondmind-labs/trieste. Issue #90. Secondmind Labs. (December 2, 2020). Retrieved 11/08/2021 from https://github.com/secondmind-labs/trieste/issues/90.

[17] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan. 2019. Boa meets Python: a Boa dataset of Data Science software in Python language. In *Mining Software Repositories*, 577–581. DOI: 10.1109/MSR.2019.00086.

[18] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, and Xin Peng. 2021. Characterizing performance bugs in Deep Learning systems. (December 3, 2021). arXiv: 2112.01771 [cs.SE].

[19] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: a tool for processing and classifying GitHub commits. In *International Symposium on Software Testing and Analysis* (ISSTA '17). ACM, 396–399. DOI: 10.1145/3092703.3098230.

[20] Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. Challenges in migrating imperative DL programs to graph execution: an empirical study. (January 21, 2022). DOI: 10.5281/zenodo.5601987.

[21] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: a flexible and efficient Machine Learning library for heterogeneous distributed systems. In *Workshop on Machine Learning Systems at NIPS*. arXiv: 1512.01274 [cs.DC].

[22] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of Deep Learning based mobile applications. In *International Conference on Software Engineering*. ACM/IEEE. IEEE. DOI: 10.1109/icse43902.2021.00068.

[23] François Chollet. 2020. *Deep Learning with Python*. (2nd edition). Manning.

[24] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding software-2.0: a study of Machine Learning library usage and evolution. *ACM Transactions on Software Engineering and Methodology*. DOI: 10.1145/3453478.

[25] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering repetitive code changes in Python ML systems. In *International Conference on Software Engineering* (ICSE '22). To appear.

[26] Vincent Dutordoir. 2020. Bug fix: compilation issue quadrature. Pull request #1418. GPflow/GPflow. Cambridge University. (April 8, 2020). Retrieved 01/14/2022 from https://github.com/GPflow/GPflow/pull/1418#issue-596552141.

[27] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *International Conference on Software Engineering*, 779–790. ISBN: 978-1-4503-2756-5.

[28] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles* (SOSP '01). ACM, Banff, Alberta, Canada, 57–72. DOI: 10.1145/502034.502041.

[29] Facebook Inc. 2019. PyTorch documentation. TorchScript. en. Retrieved 02/19/2021 from https://pytorch.org/docs/stable/jit.html.

[30] Aurélien Geron. 2019. Cryptic error message when assigning to a variable in a tf.function. Issue #30768. tensorflow/tensorflow. (July 21, 2019). Retrieved 01/17/2022 from https://github.com/tensorflow/tensorflow/issues/30768.

[31] Aurélien Geron. 2019. Surprising random seed behavior when using @tf.function. Issue #33297. (October 13, 2019). Retrieved 01/03/2022 from https://git.io/JSiac.

[32] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. 2019. On the classification of software change messages using multi-label active learning. In *Symposium on Applied Computing*. ACM/SIGAPP. ACM, 1760–1767. DOI: 10.1145/3297280.3297452.

[33] GitHub, Inc. 2021. Search. REST API Reference. GitHub Docs. Retrieved 12/02/2021 from https://docs.github.com/en/rest/reference/search.

[34] Google LLC. 2021. Better performance with tf.function. (February 4, 2021). Retrieved 02/19/2021 from https://tensorflow.org/guide/function.

[35] Google LLC. 2022. Introduction to graphs and tf.function. (January 19, 2022). Retrieved 01/20/2022 from https://tensorflow.org/guide/intro_to_graphs.

[36] Google LLC. 2021. Introduction to variables. TensorFlow core. (November 11, 2021). Retrieved 01/03/2022 from https://www.tensorflow.org/guide/variable.

[37] Google LLC. 2021. IREE: intermediate representation execution environment. (December 1, 2021). Retrieved 12/01/2021 from https://git.io/JMrPT.

[38] Google LLC. 2021. Migrate your TensorFlow 1 code to TensorFlow 2. Automatic conversion script. TensorFlow Core. (May 27, 2021). Retrieved 05/27/2021 from https://tensorflow.org/guide/migrate#automatic_conversion_script.

[39] Google LLC. 2021. Random number generation. Interaction with tf.function. TensorFlow Core. (November 16, 2021). Retrieved 01/07/2022 from https://tensorflow.org/guide/random_numbers#interaction_with_tffunction.

[40] Google LLC. 2021. tf.compat.v1.Session. (May 14, 2021). Retrieved 07/06/2021 from https://tensorflow.org/api_docs/python/tf/compat/v1/Session#run.

[41] Google LLC. 2021. tf.debugging.assert_equal. Retrieved 01/13/2022 from https://tensorflow.org/api_docs/python/tf/debugging/assert_equal#returns.

[42] Google LLC. 2021. tf.py_function. Version 2.7.0. (November 5, 2021). Retrieved 01/05/2022 from https://tensorflow.org/api_docs/python/tf/py_function.

[43] Google LLC. 2021. tf.random.set_seed. (December 4, 2021). Retrieved 01/03/2022 from https://tensorflow.org/api_docs/python/tf/random/set_seed.

[44] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *International Symposium on Empirical Software Engineering and Measurement*. DOI: 10.1145/2961111.2962602.

[45] Y. Hashemi, M. Nayebi, and G. Antoniol. 2020. Documentation of Machine Learning software. In *International Conference on Software Analysis, Evolution and Reengineering*. (February 2020). DOI: 10.1109/SANER48275.2020.9054844.

[46] Horace He. 2019. The state of Machine Learning frameworks in 2019. Retrieved 04/01/2021 from https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry.

[47] Chin Huang. 2020. Migrate to tf.module and add support for SavedModels. Pull request #603. onnx/onnx-tensorflow. IBM. (July 2, 2020). Retrieved 12/14/2021 from https://git.io/JDEoD.

[48] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in Deep Learning systems. In *International Conference on Software Engineering*. DOI: 10.1145/3377811.3380395.

[49] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on Deep Learning bug characteristics. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. (August 2019). DOI: 10.1145/3338906.3338955.

[50] Md Johirul Islam, Hoan Anh Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. What do developers ask about ML libraries? a large-scale study using Stack Overflow. (2019). arXiv: 1906.11940 [cs.SE].

[51] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: fix patterns and challenges. In *International Conference on Software Engineering*. DOI: 10.1145/3377811.3380378.

[52] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The scent of deep learning code: an empirical study. In *Mining Software Repositories*. DOI: 10.1145/3379597.3387479.

[53] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. 2019. Speculative symbolic graph execution of imperative Deep Learning programs. *SIGOPS Oper. Syst. Rev.*, 53, 1, (July 2019), 26–33. ISSN: 0163-5980. DOI: 10.1145/3352020.3352025.

[54] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Programming Language Design and Implementation*. DOI: 10.1145/2254064.2254075.

[55] Hui Jin, Kan Qiao, Xian-He Sun, and Ying Li. 2011. Performance under failures of MapReduce applications. In *International Symposium on Cluster, Cloud and Grid Computing*. DOI: 10.1109/ccgrid.2011.84.

[56] Raffi Khatchadourian. 2021. graph_execution_time_comparison.ipynb. (February 23, 2021). Retrieved 11/03/2021 from https://bit.ly/3bwrhVt.

[57] Raffi Khatchadourian and Hidehiko Masuhara. 2018. Proactive empirical assessment of new language feature adoption via automated refactoring: the case of Java 8 default methods. *The Art, Science, and Engineering of Programming*, 2, 6, 6:1–6:30, 3. DOI: 10.22152/programming-journal.org/2018/2/6.

[58] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An empirical study on the use and misuse of Java 8 streams. In *International Conference on Fundamental Aspects of Software Engineering*. ETAPS. (April 2020), 97–118. DOI: 10.1007/978-3-030-45234-6_5.

[59] David Knox. 2020. Bug in TensorFlow only allows it to run once. Issue #13. MLH-Fellowship/neuro-art. MLH Fellowship. (November 10, 2020). Retrieved 01/13/2022 from https://github.com/MLH-Fellowship/neuro-art/issues/13.

[60] Samuel Matthew Koesnadi. 2021. Fixed all . . . this should work. samuelmat19/DDPG-tf2. 02a3f29. ML6. (February 26, 2021). Retrieved 01/12/2022 from https://github.com/samuelmat19/DDPG-tf2/commit/02a3f297#r47584455.

[61] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static analysis of shape in TensorFlow programs. In *European Conference on Object-Oriented Programming*. Volume 166, 15:1–15:29. DOI: 10.4230/LIPIcs.ECOOP.2020.15.

[62] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. 2021. Detecting TensorFlow program bugs in real-world industrial environment. In *International Conference on Automated Software Engineering*. IEEE. DOI: 10.1109/ase51524.2021.9678891.

[63] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2020. Is using Deep Learning frameworks free? Characterizing technical debt in Deep Learning frameworks. In *International Conference on Software Engineering* (ICSE-SEIS '20). DOI: 10.1145/3377815.3381377.

[64] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. 2019. A taxonomy of software engineering challenges for Machine Learning systems: an empirical investigation. In *Agile Processes in Software Engineering and Extreme Programming*, 227–243. DOI: 10.1007/978-3-030-19034-7_14.

[65] Amir Makhshari and Ali Mesbah. 2021. IoT bugs and development challenges. In *International Conference on Software Engineering*. ACM/IEEE. IEEE, (May 2021). DOI: 10.1109/icse43902.2021.00051.

[66] Michał Marcinkiewicz. 2021. Deep Learning examples. NVIDIA. (March 1, 2021). Retrieved 05/05/2021 from https://git.io/J2vFG.

[67] Chirag Modi. 2021. bug boxsize=nc. modichirag/galference. af1664e. UC Berkeley. (April 16, 2021). Retrieved 01/10/2022 from https://git.io/J9ciM.

[68] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: imperative-style coding with graph-based performance. (2019). arXiv: 1810.08061 [cs.PL].

[69] Aaron Mondal. 2020. Remove @tf.function in tfa.image.equalize. Issue #2263. eomii. (December 3, 2020). Retrieved 01/10/2022 from https://git.io/J9cAb.

[70] Aaron Mondal. 2020. Remove tf.function decorator in tfa.image.equalize. eomii. (December 8, 2020). Retrieved 01/10/2022 from https://git.io/J9chg.

[71] Sean Morgan. 2019. Dense image warp tests are flaky. AWS. (April 3, 2019). Retrieved 01/13/2022 from https://github.com/tensorflow/addons/issues/138#issue-428951400.

[72] Sean Morgan. 2019. FIX: dense image warp bug. (April 17, 2019). Retrieved 01/13/2022 from https://github.com/tensorflow/addons/pull/187.

[73] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. "jumping through hoops": why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering*. ACM, (May 2016), 935–946. DOI: 10.1145/2884781.2884790.

[74] Amin Nikanjam and Foutse Khomh. 2021. Design smells in Deep Learning programs: an empirical study. In *International Conference on Software Maintenance and Evolution*. IEEE, 332–342. DOI: 10.1109/ICSME52107.2021.00036.

[75] NVIDIA Corporation. 2021. TensorRT open source software. (December 1, 2021). Retrieved 12/01/2021 from https://git.io/fjVoO.

[76] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Softw. Engg.*, 18, 6, (December 2013), 1047–1089. ISSN: 1382-3256. DOI: 10.1007/s10664-012-9236-6.

[77] Alexandre Passos. 2019. tf.function-decorated function tried to create variables on non-first call. (June 11, 2019). Retrieved 01/13/2022 from https://github.com/tensorflow/tensorflow/issues/27120#issuecomment-500975337.

[78] Adam Paszke et al. 2019. PyTorch: an imperative style, high-performance Deep Learning library. (December 3, 2019). arXiv: 1912.01703 [cs.LG].

[79] Anton Poluektov. 2021. Fit bug in Blatt-Weisskopf; update Dalitz decomposition. Apoluekt/AmpliTF. d02db12. CERN. (July 19, 2021). Retrieved 01/03/2022 from https://git.io/JSi9f.

[80] Chase Roberts. 2020. Added jitted ncon. Pull request #623. google/TensorNetwork. Xanadu. (May 26, 2020). Retrieved 01/10/2022 from https://git.io/J9cMx.

[81] Anurag Saha Roy. 2021. Performance bottleneck due to tf.function retracing. Issue #74. TensorFlow. q-optimize/c3. (March 18, 2021). Retrieved 11/08/2021 from https://github.com/q-optimize/c3/issues/74.

[82] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden technical debt in Machine Learning systems. In *Advances in Neural Information Processing Systems*, 2503–2511.

[83] Paulo Silveira, Umme Ayda Mannan, Eduardo Santana Almeida, Nachi Nagappan, David Lo, Pavneet Singh Kochhar, Cuiyun Gao, and Iftekhar Ahmed. 2021. A deep dive into the impact of COVID-19 on software development. *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2021.3088759.

[84] Stack Exchange Inc. 2020. Should I use @tf.function for all functions? Stack Overflow. (January 21, 2020). Retrieved 11/08/2021 from https://stackoverflow.com/questions/59847045/should-i-use-tf-function-for-all-functions.

[85] Eric Stavarache. 2019. tf.function-decorated function tried to create variables on non-first call. Issue #27120. ETH Zürich. (March 25, 2019). Retrieved 01/13/2022 from https://github.com/tensorflow/tensorflow/issues/27120.

[86] Tzu-Wei Sung. 2020. Remove tf.function decorator in tfa.image.equalize. (December 5, 2020). Retrieved 01/10/2022 from https://git.io/J9Cft.

[87] Florian Tambon, Amin Nikanjam, Le An, Foutse Khomh, and Giuliano Antoniol. 2021. Silent bugs in Deep Learning frameworks: an empirical study of Keras and TensorFlow. (2021). arXiv: 2112.13314 [cs.SE].

[88] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An empirical study of refactorings and technical debt in Machine Learning systems. In *International Conference on Software Engineering*, 238–250. DOI: 10.1109/ICSE43902.2021.00033.

[89] Yiming Tang, Allan Spektor, Raffi Khatchadourian, and Mehdi Bagherzadeh. 2022. Automated evolution of feature logging statement levels using Git histories and degree of interest. *Science of Computer Programming*, 214, (February 1, 2022), 102724. DOI: 10.1016/j.scico.2021.102724.

[90] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in Machine Learning systems. In *International Symposium on Software Reliability Engineering*. DOI: 10.1109/ISSRE.2012.22.

[91] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in C. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 752–762. DOI: 10.1145/3106237.3106300.

[92] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2020.3007722.

[93] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *Mining Software Repositories*. IEEE, (May 2017), 413–424. DOI: 10.1109/MSR.2017.59.

[94] Jun Xu. 2021. Deadlock on recursive tf.function-decorated function. Issue #35540. (October 8, 2021). Retrieved 01/05/2022 from https://git.io/JSS4P.

[95] Mac Xu. 2019. Calling tf.function from tf.py_function in dataset.map hangs. (September 11, 2019). Retrieved 01/05/2022 from https://git.io/JSSBw.

[96] H. Yamada. 2020. Reduce tf.function retracing. Pull Request #100. (August 10, 2020). Retrieved 11/08/2021 from https://github.com/keiohta/tf2rl/pull/100.

[97] Jiang Yi. 2022. Remove 'experimental_relax_shapes=true' to avoid none shape. University of Chinese Academy of Sciences. (January 4, 2022). Retrieved 01/10/2022 from http://github.com/jiangyi15/tf-pwa/commit/0db1#r62889998.

[98] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing Deep Learning applications. In *International Symposium on Software Reliability Engineering*. (October 2019). DOI: 10.1109/ISSRE.2019.00020.

[99] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *International Symposium on Software Testing and Analysis*. DOI: 10.1145/3213846.3213866.

[100] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. 2021. Unveiling the mystery of API evolution in Deep Learning frameworks: a case study of TensorFlow 2. In *International Conference on Software Engineering* (ICSE-SEIP). DOI: 10.1109/ICSE-SEIP52600.2021.00033.

[101] Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. 2020. HARP: holistic analysis for refactoring Python-based analytics programs. In *International Conference on Software Engineering*, 506–517. DOI: 10.1145/3377811.3380434.