Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks

Jiakun Liu*†
Zhejiang University
College of Computer Science and Technology
Hangzhou, Zhejiang, China
jkliu@zju.edu.cn

Qiao Huang
Zhejiang University
College of Computer Science and Technology
Hangzhou, Zhejiang, China
tkdsheep@zju.edu.cn

Xin Xia‡
Monash University
Faculty of Information Technology
Melbourne, Victoria, Australia
Xin.Xia@monash.edu

Emad Shihab
Concordia University
Department of Computer Science and Software Engineering
Montreal, Canada
eshihab@encs.concordia.ca

David Lo
Singapore Management University
School of Information System
Singapore
davidlo@smu.edu.sg

Shanping Li
Zhejiang University
College of Computer Science and Technology
Hangzhou, Zhejiang, China
shan@zju.edu.cn

ABSTRACT

Developers of deep learning applications (shortened as application developers) commonly use deep learning frameworks in their projects. However, due to time pressure, market competition, and cost reduction, developers of deep learning frameworks (shortened as framework developers) often have to sacrifice software quality to satisfy a shorter completion time. This practice leads to technical debt in deep learning frameworks, which results in the increasing burden to both the application developers and the framework developers in future development.

In this paper, we analyze the comments indicating technical debt (self-admitted technical debt) in 7 of the most popular open-source deep learning frameworks. Although framework developers are aware of such technical debt, typically the application developers are not. We find that: 1) there is a significant number of technical debt in all the studied deep learning frameworks. 2) there is design debt, defect debt, documentation debt, test debt, requirement debt, compatibility debt, and algorithm debt in deep learning frameworks. 3) the majority of the technical debt in deep learning framework is design debt (24.07% - 65.27%), followed by requirement debt (7.09% - 31.48%) and algorithm debt (5.62% - 20.67%). In some projects, compatibility debt accounts for more than 10%. These findings illustrate that technical debt is common in deep learning frameworks, and many types of technical debt also impact the deep learning applications. Based on our findings, we highlight future research directions and provide recommendations for practitioners.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Self-admitted Technical Debt, Deep Learning, Categorization, Empirical Study

ACM Reference Format:
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 *Software Engineering in Society (ICSE-SEIS'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377815.3381377

1 INTRODUCTION

Developers of deep learning applications (shortened as application developers) usually use deep learning frameworks, such as TensorFlow[1], Keras[2] and Deeplearning4j (shortened as DL4J)[3], to implement their deep learning models in their projects. By using deep learning frameworks, application developers are freed from the concrete implementation of task scheduling (such as control flow and data transfer between tasks), specific algorithms (such as Batch Normalization[4], Max Pooling[5]), and so on, and pay more attention to their application development.

Developers of deep learning framework projects (shortened as framework developers) are expected to deliver high-quality products or services continuously. However, due to time pressure, market competition and cost reduction [16], framework developers are often confronted with a dilemma: a shorter completion time

*Also with Zhejiang University, Ningbo Research Institute.
†Also with PengCheng Laboratory.
‡Corresponding author

ICSE-SEIS'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7125-4/20/05…$15.00
https://doi.org/10.1145/3377815.3381377

[1]https://github.com/tensorflow/tensorflow
[2]https://github.com/keras-team/keras
[3]https://github.com/deeplearning4j/deeplearning4j
[4]https://docs.microsoft.com/en-us/cognitive-toolkit/batchnormalization
[5]https://keras.io/layers/pooling/

or better software quality. Compromised decisions lead to the increasing burden in the future development life cycle. The metaphor, technical debt, first proposed by Cunningham in 1993 [4], describes such decisions. Previous research found that technical debt is detrimental, e.g., increasing the cost, negatively impacting the product quality [31], [33], [6]. However, it is still unclear if technical debt exists in deep learning frameworks and what is its impact.

To help framework developers manage projects and application developers use deep learning frameworks, in this paper, we investigate the technical debt in deep learning frameworks. Previous research found that most of the developers do not consider technical debt as the result of sloppy programming or poor developer discipline. Instead, they consider it as a result of intentional decisions to trade off competing concerns during development [12]. Therefore, to identify technical debt in deep learning frameworks, we employ the self-admitted technical debt (i.e., SATD) as an indicator of technical debt. Such technical debt is the *comment* that is intentionally introduced by developers to alert the inadequacy of the solution [23]. For example, in the open-source project TensorFlow, a comment saying *TODO (b/26910386): Identify why this infrequently causes timeouts*, indicates that the corresponding code is problematic and needs further investigation. Framework developers are aware of such technical debt, but application developers may not be. This practice may cause applications to be negatively impacted or limit their compatibilities.

We mine the SATD in 7 of the most popular open-source deep learning frameworks by manually identifying and classifying 234,260 comments in the latest stable version. With these data, we explore several questions:

(1) **Is technical debt prevalent in deep learning frameworks?**
We find that all these 7 deep learning frameworks have technical debt. More specifically, TensorFlow has the largest amount of technical debt (3,775 instances) while Keras has the smallest amount of technical debt (54 instances).

(2) **What types of technical debt exist in deep learning frameworks?**
We find that besides the five categories, i.e., design debt, defect debt, documentation debt, requirement debt and test debt, which have been observed in non-deep learning projects, compatibility debt and algorithm debt are also discovered in deep learning frameworks. This indicates that compared with developers in non-deep learning projects, developers in deep learning frameworks are confronted to new challenges.

(3) **What is the distribution of different types of technical debt in deep learning frameworks?**
We find that the majority of technical debt is the design debt (24.07% - 65.27%), followed by requirement debt (7.09% - 31.48%) and algorithm debt (5.62%–20.67%). In some projects, compatibility debt accounts for more than 10%, which cannot be ignored.

Based on our findings, we provide actionable suggestions for practitioners and researchers. For example, if the deep learning application needs to use the out-of-box implementation of the cutting-edge deep learning algorithms provided by deep learning frameworks, we suggest the application developers not to use the frameworks with more requirement debt, such as DL4J, where many algorithms remained un-implemented. For deep learning researchers, to mitigate the risks introduced by deep learning frameworks, we suggest they should evaluate their algorithms on more deep learning frameworks rather than only based on the out-of-box implementation provided by one deep learning framework.

2 RELATED WORK

We divide our related work into two parts: the research works on technical debt and the research works on software engineering for deep learning.

2.1 Technical Debt

Cunningham [4] introduced the metaphor, technical debt, to describe the consequences of poor software development. After the proposal of the metaphor, many researchers focus on how technical debt has been used to communicate the issues that developers find in the code in a way that managers can understand [26], [14] [3].

Potdar and Shihab [23]'s work used comments to identify technical debt and nominates such technical debt as **self-admitted technical debt** (i.e., SATD). They find that SATD is common cross projects and 2.4-31% of the files in four non-deep learning projects contain such debt. Bavota and Russo [2] replicate the study of Potdar and Shihab [23]'s work on a large set of non-deep learning projects, i.e., Apache and Eclipse projects and find that approximately 57% of SATDs get removed and around 63% of SATD are self-removed. Maldonado et al. [17]'s work inspects the introduction and removal of SATD in five open source non-deep learning projects and find that the majority of SATD is removed (74.4% on average) in 82 - 613.2 days on average.

Many previous works also study the negative impact of technical debt during the development of projects. Zazworka et al. [34]'s work conducts a study to measure the impact of technical debt on software quality. They find that god classes are more likely to change, and therefore, have a higher impact in software quality. Fontana et al. [6]'s work investigates design technical debt appearing in the form of code smells, namely god classes, data classes and duplicated code. They propose an approach to classify which one of the different code smells should be addressed first, based on a risk scale. Wehaibi et al. [32]'s work finds that SATD leads to more complex changes in the future development process. Kamei et al. [10]'s work proposes a method to measure technical debt interest using SATD in the source code. They find that around 42% of the technical debt in the studied projects incurs more burdens on developers.

Moreover, there have been many studies on the categories of SATD as well. Alves et al. [1]'s work proposes an ontology on technical debt terms. They gather definitions and indicators of technical debt that were scattered across the literature and find several different categories of technical debt, e.g., architecture debt, build debt, code debt, design debt, defect debt, etc. Besides, Zazworka et al. [35]'s work and Li et al. [15]'s work also illustrate the categories of technical debt which are found in their studied projects. The work most relevant to us is Maldonado and Shihab [18]'s work, where they manually analyze the comments of 5 open source non-deep learning projects. They find that there are five categories of technical debt that are admitted in comments: design debt, defect debt, documentation debt, requirement debt and test debt.

Compared with these research works, our research identifies and investigates technical debt in a specific set of projects: deep learning frameworks, which not only have distinct characteristics, but also have widespread impacts on various deep learning applications that use them.

2.2 Software Engineering for Deep Learning

Former researchers focus on the test of deep learning projects. Pei et al. [22] propose DeepXplore to systematically test DL systems and automatically identify erroneous behaviors without manual labels. Tian et al. [30] propose DeepTest to automatically test DNN-driven autonomous cars, which can use test images that generated by different realistic transformations like rain, fog and lighting conditions.

Besides works focusing on testing of deep learning projects, Zhang et al. [36]'s work studies the characteristics of deep learning defects. They study TensorFlow application bugs from StackOverflow and Github, and find the root causes of the defects, e.g., incorrect model parameter or structure. Moreover, Sculley et al. [25] empirically summarize the technical debt in machine learning systems during their development. They explore several ML-specific risk factors in deep learning project design, including boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, and so on.

Different from the prior research works, our work inspects deep learning frameworks from another perspective: the comments indicating technical debt. As we indicate in Section 2.1, such technical debt is detrimental. Although framework developers are aware of such technical debt, typically the application developers are not.

3 CASE STUDY SETUP

In this section, we describe the steps that we took for project selection, project data extraction, comments extraction, manual classification of the comments in the latest stable version.

3.1 Project Selection

We focus on open-source deep learning frameworks hosted on Github. We exclude deep learning systems that build upon such frameworks, or general-purpose mathematical libraries that those deep learning frameworks build upon.

To do so, we first search repositories labeled as *deeplearning* and *deep learning* topics[6] in GitHub. Then, we identify the deep learning framework projects by reading the readme file of the projects. As a result, we include 7 deep learning frameworks with the largest number of stars that are written in 3 programming languages (C++, Python and Java) as subject frameworks for our study. They include: TensorFlow, Keras, Deeplearning4j (shortened as DL4J), Caffe[7], PyTorch[8], MXNet[9] and Microsoft Cognitive Toolkit (known as CNTK)[10]. Table 1 provides statistics about each framework in our study, including the release we used, the total number of lines of code, and the main programming languages. Following the previous study by Maldonado and Shihab [18], we calculate the total number of code lines using SLOCCount[11].

Table 1: Summary of the Studied Projects. For each framework, we present the release we used, the total number of lines of code, and the main programming languages.

Framework	Release	#Lines of Code	Languages
TensorFlow	v1.9.0	1,821,016	Python, C++
Keras	2.2.2	42,182	Python
Caffe	1.0	76,322	C++
PyTorch	v0.4.0	617,255	Python, C++
MXNet	1.2.1	305,755	Python, C++
CNTK	v2.5.1	324,472	Python, C++
DL4J	0.9.1	361,366	Java

3.2 Comment Extraction

We need the comments in the latest stable version. To discriminate between source code and comment lines, we use the srcML Toolkit[12], which is capable of parsing source files which are coded by C++, Java and so on except Python, into XML files. Then, we develop a Java-based tool which parses the XML files produced by srcML and records the file name, class name, method name, and comment content while traversing the DOM tree. For Python files which is not supported by srcML, we utilize the tokenize module[13] in the Python standard library, which provides a lexical scanner for Python source code to identify all comments. Finally, we extract a total of 234,260 comments in the latest stable version.

3.3 Manual Classification of the Comments in the Latest Stable Version

We have two goals in the process of manual classification: find comments indicating technical debt from all comments in the latest stable version, and classify the identified technical debt instances into different categories. We utilize the categories which are found in Maldonado and Shihab [18]'s work as a starting point, where they analyze the comments of 5 non-deep-learning open source projects and find that the technical debt in non-deep learning projects can be classified into five categories: design debt, defect debt, documentation debt, requirement debt, and test debt.

In practice, we perform three iterations of a card sorting approach [29] to classify the 234,260 deep learning frameworks comments in the latest stable version. Concretely, in the first iteration of classification, we try to ensure that our classification standard is consistent with previous work. To do so, we first randomly pick 100 comments from the dataset provided by Maldonado and Shihab [18]'s work, then the first two authors manually classify these sentences according to Maldonado and Shihab [18]'s work. A discussion on the disagreements with Maldonado and Shihab [18]'s work is performed after the classification process. To validate our classification standard, the first author selected another 500 comments from the dataset provided by Maldonado and Shihab [18]'s work and manually classified them. Then, we calculate the Cohen's kappa coefficient [20] and obtain a result of +0.85, which indicates

[6] https://blog.github.com/2017-01-31-introducing-topics/
[7] https://github.com/BVLC/caffe
[8] https://github.com/pytorch/pytorch
[9] https://github.com/apache/incubator-mxnet
[10] https://github.com/Microsoft/CNTK
[11] https://dwheeler.com/sloccount/
[12] https://www.srcml.org/
[13] https://docs.python.org/3/library/tokenize.html

a high level of agreement with the classification given by the first author and Maldonado and Shihab [18]'s work.

In the second classification iteration, we first randomly pick 2,000 comments from all the 234,260 comments in the latest stable version of these 7 deep learning frameworks. The first two authors separately manually classify these comments into six categories, i.e., the "NON-technical debt" category and the five categories which are found in Maldonado and Shihab [18]'s work. If an author considers the sentence could not be classified into any of the six categories, it is set aside for further discussion. Then, the first two authors discuss the disagreements in the classification process and the cases that could not be categorized into any one of five categories. As a result, we have the following two findings:

(1) A large amount of comments in our selected dataset express that the current *algorithm* of deep learning model, neural network module and computation function are sub-optimal, which impacts the performance of the system. However, the categories proposed in earlier work cannot cover this kind of technical debt. To better understand this case, we propose algorithm debt to enrich the technical debt categories proposed in earlier work.
(2) A large amount of comments in our selected dataset express that the implementation which is based on the current external dependencies is a workaround. However, though the current external dependencies are still immature or buggy, there is no better choice temporarily. However, this kind of technical debt cannot be found in Maldonado and Shihab [18]'s work. To better apperceive this case, we propose compatibility debt to enhance the technical debt categories proposed in earlier work.

In the third classification iteration, the first author classifies the remaining comments. Then, we select another 2,000 comments from the 234,260 comments and invite an independent Ph.D. student, who is not an author of this paper, to manually classify them. A high level of agreement between the classification given by the two different students is reported with Cohen's kappa coefficient of +0.79, which is considered to be high agreement [5].

4 TECHNICAL DEBT IN DEEP LEARNING FRAMEWORKS

In this section, we present the result of our experiments in the following subsections. Concretely, we present the number of technical debt at different granularities, the categories of technical debt in deep learning frameworks, and their corresponding proportions.

4.1 RQ1: Is technical debt prevalent in deep learning frameworks?

Motivation: As the first work on technical debt in deep learning frameworks, identification and quantification of technical debt instances in the latest stable version enables us to have basic knowledge of how common technical debt is in deep learning framework projects. To better understand the prevalence of technical debt in deep learning frameworks, we also would like to quantify the number of files/ classes/ methods that contain technical debt instances.

Approach: We first quantify the number of technical debt instances that are found in Section 3.3. Then, we use the meta information of the technical debt instances, e.g., the file/ class/ method that contain the technical debt instance, which are obtained in Section 3.2, to find the SATD related entity at different granularities. As is indicated in Section 3.1, the size of different projects are different. We normalize the result by the number of total files, classes or methods that the project has respectively.

Table 2: SATD Distribution at Different Granularities. For each project, we present the total number of comments/ methods/ classes/ files, the number of comments/ methods/ classes/ files that indicate or contain technical debt, and their corresponding proportion.

Granularity	Project Name	#Total	#SATD	%
Comment	TensorFlow	131,238	3,775	2.87%
	Keras	2,986	54	1.80%
	Caffe	5,901	201	3.40%
	PyTorch	24,449	976	3.99%
	MXNet	20,178	236	1.16%
	CNTK	34,150	1,443	4.22%
	DL4J	15,358	474	3.08%
Method	TensorFlow	64,066	1,392	2.17%
	Keras	2,388	32	1.34%
	Caffe	2,104	39	1.85%
	PyTorch	17,380	435	2.50%
	MXNet	7,552	101	1.33%
	CNTK	9,095	246	2.70%
	DL4J	12,068	108	0.89%
Class	TensorFlow	7,285	647	8.88%
	Keras	198	8	4.04%
	Caffe	305	13	4.26%
	PyTorch	1,684	122	7.24%
	MXNet	712	24	3.37%
	CNTK	341	49	14.36%
	DL4J	1,842	127	6.89%
File	TensorFlow	7,600	1,608	21.15%
	Keras	178	28	15.73%
	Caffe	354	56	15.81%
	PyTorch	2,861	482	16.84%
	MXNet	1,149	153	13.31%
	CNTK	1,238	345	27.86%
	DL4J	1,588	212	13.35%

Result: Table 2 shows the prevalence of technical debt in deep learning frameworks at different levels of granularity. As a result, in terms of the number of the technical debt instances, we find that there are a total of 7,159 comments that indicate technical debt, which corresponds to 2.93% of all comments. Concretely, TensorFlow has the largest amount of technical debt instances (3,775) that are admitted by developers, while there are only 54 technical debt instances that are admitted by developers in Keras. CNTK has the largest proportion (4.22%) among all studied projects while only 1.16% comments indicate technical debt in MXNet.

Moreover, 13.35–27.86% of files, 3.37%–14.36% of classes, 0.89–2.70% of methods/functions contain one or more instances of SATD. More specifically, there is a sharp percentage drop from file level to class level. One reason is some comments that are used as class/file declaration lie out of the range of class. Besides, in deep learning framework projects, the widespread use of scripting languages (such as Python), which are not class based also intensifies to this case.

In fact, at any granularities, CNTK contains the most percentage of SATD instances while MXNet contains the least.

This result reveals that technical debt is common in deep learning frameworks, especially in Tensorflow and CNTK. We will compare the technical debt in deep learning frameworks with that in non-deep learning projects in Section 5.1. During the development of deep learning frameworks, framework developers also have unresolved technical problems in their projects. They are aware of these technical problems and write down them in the comments as admitted technical debt. However, the application developers do not know the existence of the technical debt and use these frameworks with risks.

We find that technical debt is prevalent in deep learning framework projects. Concretely, 13.35 - 27.86% of files, 3.37% - 14.36% of classes, 0.89 - 2.70% of methods/functions contain SATD.

4.2 RQ2: What types of technical debt exist in deep learning frameworks?

Motivation: Although we have identified technical debt instances in deep learning frameworks, it is still unclear that what types of technical debt exist in deep learning frameworks. Considering that different categories of admitted technical debt indicate different types of problems that are faced by framework developers, different categories of technical debt can also have different impacts on both framework developers and application developers. Therefore, we wish to clarify the categories of technical debt in deep learning frameworks and their possible impact on the development of deep learning frameworks and applications.

Approach: We present the results of the manual classification process presented in Section 3.3. To better illustrate different categories of technical debt, we also present several examples for each technical debt category, as well as its possible impacts on framework developers and application developers

Result: We find that the following categories of technical debt in deep learning frameworks:

1. Design debt indicates sub-optimal design, e.g., misplaced code, lack of abstraction, long methods, poor implementation, workarounds, or temporary solutions on the usage of other internal functions.

We find that in deep learning frameworks, some of design debt instances happen because **design details of the framework have not been determined yet when the code is implemented**. For example:

> *"TODO: this option should be abstracted, if we decide to generalize this trainingmaster"* - [from DL4J][14]
> *"TODO: make it public?"* - [from Keras][15]

Such comments illustrate that although the current implementation has satisfied the functionalities that are explicit in the requirement, the design of the code is sub-optimal. Usually, framework developers report their doubts or provide suggestions about the design of code in such comments that indicate technical debt. We suggest that the project manager of the frameworks should determine or respond to the details of the code design as quickly as possible.

Moreover, we observe that **poor implementation of code** is one of the reasons that lead to design debt, which represents a compromise between code quality and specific functional or non-functional requirements. For example:

> *"TODO(kenton@google.com): There are other ways to get the time on Windows, like GetTickCount() or GetSystemTimeAsFileTime(). MinGW supports these. consider using them instead."* - [from Caffe][16]

Usually, such technical debt is because the developers who implement related code are not familiar with the code. We expect the developers with more experience in deep learning frameworks to update the poor implementation.

We also observe that **the violation of object-oriented design principles** is another reason that leads to design debt, which can **lead to more changes on related classes** [34]. For example:

> *"TODO(b/32239616): This kernel should be moved into Eigen and vectorized."* - [from TensorFlow][17]

The TensorFlow developer admits there is a misplaced code. This practice leads to the responsibilities of the *Eigen* to be spread across more than one class. When there are new requirements to change the behaviors of *Eigen*, all classes related to class *Eigen* have to be changed, which leads to the instability of the framework. This is a violation of the Single Responsibility Principle[18].

We observe that **duplicated code** is a reason that leads to design debt as well, which can **cause an increase in maintenance costs** [8], [7]. For example:

> *"TODO(andydavis) Remove some of the code duplicated between this module and that in 'common_runtime/function.cc'. A few string constant used throughout this module."* - [from TensorFlow][19]
> *"TODO(shelhamer) loss normalization should be pulled up into LossLayer, instead of duplicated here and in SoftMaxWithLossLayer"* - [from Caffe][20]

These developers admit that there is duplicated code. Duplicated code is detrimental because (1) the update of code have to be performed in multiple places [13], [24] and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [9]. The existence of duplicated code in deep learning frameworks indicates that copy and paste activity is common during its development, and framework developers are aware of such activity.

2. Defect debt corresponds to code that behaves in unintended ways, and developers postpone repair because of additional factors (e.g., the complexity of repair or time pressure).

We observe that **the use of widely adopted probability and statistics-based algorithm** can often lead to defect debt. For example:

[14] deeplearning4j/deeplearning4j-scaleout/spark/dl4j-spark-parameterserver/src/main/java/org/deeplearning4j/spark/parameterserver/training/SharedTrainingMaster.java
[15] keras/keras/preprocessing/sequence.py

[16] caffe/src/gtest/gtest-all.cpp
[17] tensorflow/tensorflow/core/kernels/cwise_ops.h
[18] https://en.wikipedia.org/wiki/Single_responsibility_principle
[19] tensorflow/tensorflow/core/graph/gradients.cc
[20] caffe/src/caffe/layers/sigmoid_cross_entropy_loss_layer.cpp

"Linear weights do not follow the column name. But this is a rare use case, and fixing it would add too much complexity to the code." - [from Tensorflow][21]

'TODO(phawkins): too small a maximum tensor size could lead to an infinite loop here." - [from Tensorflow][22]

The probability and statistics-based algorithms are sensitive to input data [28], [21]. These algorithms lead to the deep learning frameworks to solve problems non-deterministically and make the defects of deep learning frameworks difficult to reproduce. Framework developers usually postpone to repair the defect because it is usually a rare case, and the fix of the bug requires extra code complexity.

We also observe that **the delay caused by the need to collaborate other development teams to resolve the defects** is another reason that leads to defect debt. For example:

"TODO(Patric): lrn_within_channel will cause core dump in MKLDNN backward. Need to confirm with MKLDNN team and fix later" - [from MXNet][23]

Defect debt can lead to unstable data dependencies in applications. The accumulation of the defect debt leads the framework developers to have to process their further development based on problematic code, which negatively impacts the quality of the frameworks. Moreover, for application developers, they usually train, validate and test their model with the data processed by the deep learning frameworks. They acknowledge the bugs of the frameworks by checking issue tracking systems, such as *Issue*[24] in Github. However, some defects are only known to the framework developers, and they postpone the fix and leave these defects in frameworks as defect debt. When application developers use these flawed frameworks and trigger these bugs, defect debt can cause unexpected results in their applications. If the application developers are not aware that the incorrect behaviors are caused by the frameworks rather than their algorithmic logic, they may modify their original deep learning algorithms and expect to get the correct results based on the frameworks with defect debt. In the future, when the defect debt in frameworks is resolved, or they move to a new framework without defect debt, the models they trained with defect debt before would fail, which leads to the unstable data dependencies. In fact, many application developers complain about unstable data dependencies in deep learning frameworks during their development[25,26]. This finding suggests researchers and developers should not only notice the bugs that are reported to the issue tracking systems but also the defect debt that is identified by developers themselves and admitted in comments.

3. Documentation debt indicates missing, inadequate or incomplete documentation that explains the corresponding part of the program.

We observe that one of the reasons that lead to documentation debt is that some framework developers expect other developers or experts to provide a clear description of the code. For example:

"TODO(sibyl-vie3Poto): Write up a doc with concrete derivation and point to it from here." - [from TensorFlow][27]

"Given a numerical function "f", returns another numerical function "g", such that if "f" takes N inputs and produces M outputs, "g" takes N + M inputs and produces N outputs. I.e., if $(y_1, y_2, ..., y_M) = f(x_1, x_2, ..., x_N)$*, g is a function which is* $(dL/dx_1, dL/dx_2, ..., dL/dx_N) = g(x_1, x_2, ..., x_N, dL/dy_1, dL/dy_2, ..., dL/dy_M)$*, where L is a scalar-value function of* $(...x_i...)$*. TODO(zhifengc): Asks math expert to say the comment again."* - [from TensorFlow][28]

In the above examples, a TensorFlow developer expects the detailed mathematical derivation of the *ComputeUpdatedDual* method in *HingeLossUpdater* class, and another TensorFlow developer expects an expert's confirmation on the documentation of a specific algorithm. One possible reason is that it is difficult for framework developers not only to master the project but also to understand a diverse set of knowledge, such as the basic neural network structures and high-performance computation algorithms.

Documentation debt can lead to framework developers not fully understand collaborator's work. Documentation acts as the communication medium between members of the development team and probably the application developers. Documentation debt increases the difficulties for maintenance [11].

4. Requirement debt indicates *incompleteness* of the method, class or program at a particular time, which may mean that the original planned completion of the task exceeds the development schedule. It can also correspond to cases when new requirements are identified during the development of existing requirements but cannot be considered due to time pressure or other constraints. For example:

"TODO setup for RNN" - [from DL4J][29]

"TODO: extend this method to handle bidirection LSTMs." - [from CNTK][30]

Requirement debt can lead the application developers to have to implement concrete functions by themselves. The research of deep learning algorithms progresses at a rapid pace. Many new algorithms are still proposed nowadays, which can be applied to many areas and play an important role. Application developers expect the deep learning frameworks to provide the out-of-box implementation of these cutting-edge algorithms. However, many frameworks do not implement these algorithms in time, which makes them at a disadvantage in the fiercely competitive market. Requirement debt can lead the application developers to not be aware of the incompleteness of the methods, and they use these methods as normal and result in unexpected result[31,32]. To deal with requirement debt, application developers have to implement these algorithms by themselves.

[21] tensorflow/tensorflow/python/feature_column/feature_column_test.py
[22] tensorflow/tensorflow/compiler/tests/randomized_tests.cc
[23] incubator-mxnet/src/operator/nn/mkldnn/mkldnn_lrn-inl.h
[24] https://help.github.com/en/articles/about-issues
[25] https://hackernoon.com/how-tensorflows-tf-image-resize-stole-60-days-of-my-life-aba5eb093f35
[26] https://www.twosigma.com/insights/article/a-workaround-for-non-determinism-in-tensorflow/
[27] tensorflow/tensorflow/core/kernels/hinge-loss.h
[28] tensorflow/tensorflow/core/common_runtime/function.h
[29] deeplearning4j/deeplearning4j-nn/src/main/java/org/deeplearning4j/nn/params/BatchNormalizationParamInitializer.java
[30] CNTK/Source/CNTKv2LibraryDll/proto/onnx/CNTKToONNX.cpp
[31] https://discuss.mxnet.io/t/not-implemented-for-use-with-gpus/1093
[32] https://github.com/Microsoft/CNTK/issues/2441

5. Test debt indicates the need for improvements to address deficiencies in the test suite. We observe that some of test debt indicate that there is **insufficient test** in deep learning frameworks, which can **lead to the un-covered defects.** For example:

"TODO(fchollet): insufficiently tested." - [from TensorFlow][33]

For the above example, the TensorFlow developers admits that the current test is insufficient, which may cause bugs to remain hidden. This practice can downgrade the project quality.

We also observe that there is **dead test case** in deep learning frameworks. Below is another example of the test debt, where the TensorFlow developer admits there is dead test:

"TODO: these flags no longer exist, this test probably no longer applies" - [from CNTK][34]

Sufficient testing is good for the project, but the dead tests can lead to static analysis tools (Clover[35], IntelliJ[36]) to be unable to find abandoned source code. The legacy of abandoned source code can lead to catastrophic consequences. One example is that Knight Capital lost $465 million in 45 minutes because of unexpected behavior from dead code [25], [27]. In deep learning projects, developers usually implement new algorithms with higher accuracy. If there is dead code which is the implementation of an abandoned algorithm and is called by developers accidentally, the loss of precision could cause catastrophic consequences. For example, for self-driving systems, a sudden decrease in the accuracy can lead to an accident, and the failure of the facial recognition systems used in the bank may lead to unauthorized access to bank accounts.

6. Compatibility debt refers to debt related to a project's immature dependencies on other projects, which cannot supply all qualified services, and the current implementation is a temporary workaround.

Compatibility debt can lead to more code changes. For example:

"Moved to common.cpp instead of including boost/thread.hpp to avoid a boost/NVCC issues (#1009, #1010) on OSX. Also fails on Linux with CUDA 7.0.18." - [from Caffe][37]

"TODO(wan): report the reason of the failure. We don't do it for now as: 1. There is no urgent need for it. 2. It's a bit involved to make the errno variable thread-safe on all three operating systems (Linux, Windows, and Mac OS). 3. To interpret the meaning of errno in a thread-safe way, we need the strerror_r() function, which is not available on Windows." - [from Caffe][38]

In the above examples, the first Caffe developer complains about a boost/NVCC issue (#1009, #1010) on OSX, the second Caffe developer writes down his explanation on the postponed repairment of defect debt. The underlying dependency, i.e., CUDA 7.0.18, and operating systems, i.e., Linux, Windows, and Mac OS, in the above example, cannot provide qualified services, which leads to a workaround. This indicates that developers admit that deep learning frameworks bind themselves tightly with other open-source packages, libraries, and platforms, and freeze the project to the specific version of the dependencies. If framework developers use too many dependencies, they are also exposed to the risks brought by these dependencies[39,40].

Moreover, framework developers have to pay additional attention to the update of the dependencies in their future work. Whenever there is an updated version of the package, it needs to be re-integrated into the internal code-base. If additional custom modifications need to be made, someone needs to maintain this package internally, and they often need to be committed back to the open-source (to maintain future compatibility). This practice may require jumping through administrative hoops.

7. Algorithm debt corresponds to sub-optimal implementations of algorithm logic in the deep learning framework.

Algorithm debt can pull down the performance of the systems. Below are two examples of algorithm debt, where the Caffe developer expects a faster way to do pooling in the channel-first case and the TensorFlow developer admits that the *kCostPerUnit* needs to be optimized:

"TODO(Yangqing): Is there a faster way to do pooling in the channel-first case?" - [from Caffe][41]

"TODO(zakaria): optimize kCostPerUnit" - [from TensorFlow][42]

It is often the case that, the computation task of deep learning projects is quite heavy and for such cases sub-optimal implementation corresponding to algorithm debt can have an adverse impact.

Besides the five categories, i.e., design debt, defect debt, documentation debt, requirement debt and test debt, which have been observed in non-deep learning projects, compatibility debt and algorithm debt are also discovered in deep learning frameworks.

4.3 RQ3: What is the distribution of different types of technical debt in deep learning frameworks?

Motivation: Although we have identified different categories of technical debt in deep learning frameworks, it is still unclear that which categories of technical debt are the most common. By doing so, we can acknowledge which categories of technical debt are most confronted with the frameworks developers and can cause larger impact on application developers.

Approach: We count the number of different categories of technical debt, which is identified in Section 3.3. Table 3 presents the distribution of different categories of technical debt in different frameworks, as well as the total number of introduced SATD instances for each project. To better view the differences between different categories of technical debt, we highlight the top three categories in term of the proportion in each project in bold.

Result: As a result, we find that in most projects except Keras, design debt is the most common debt, ranging from 24.07% (in Keras) to 65.27% (in CNTK). This finding indicates that developers in deep learning frameworks show their explicit dissatisfactions

[33] tensorflow/tensorflow/python/keras/backend_test.py
[34] CNTK/Tests/UnitTests/MathTests/TensorTestsHelper.h
[35] https://www.atlassian.com/software/clover
[36] https://www.jetbrains.com/help/idea/code-coverage.html
[37] caffe/include/caffe/common.hpp
[38] caffe/src/gtest/gtest-all.cpp
[39] https://github.com/keras-team/keras/issues/3431
[40] https://stackoverflow.com/q/45984253/
[41] caffe/src/caffe/layers/pooling_layer.cpp
[42] tensorflow/tensorflow/contrib/layers/kernels/sparse_feature_c-ross_kernel.cc

Table 3: Category Distribution of Identified Technical Debt per Framework. We highlight the top 3 most common categories of each project in bold.

	TensorFlow	Keras	Caffe	PyTorch	MXNet	CNTK	DL4J
%design	**57.03%**	**24.07%**	**48.75%**	**59.73%**	**63.13%**	**65.27%**	**55.90%**
%requirement	**17.56%**	7.40%	5.62%	**12.80%**	**11.01%**	7.41%	**20.67%**
%algorithm	7.09%	**31.48%**	10.00%	**10.45%**	**10.16%**	**10.53%**	**13.92%**
%compatibility	3.78%	**35.18%**	**11.87%**	7.37%	2.96%	3.95%	0.21%
%documentation	1.27%	0.00%	**15.62%**	0.81%	0.42%	0.83%	0.42%
%test	**7.70%**	0.00%	3.12%	4.61%	3.81%	4.50%	0.63%
%defect	5.53%	1.85%	5.00%	4.20%	8.47%	**7.48%**	8.22%

on the design of current implementation the most, and they record them and expect improvement.

And following that, for most projects, either requirement debt (5.62% - 20.67%) or algorithm debt (7.09% - 31.48%) takes the second place. This finding indicates that with the rapid advancement of the cutting-edge deep learning algorithms, developers have not implemented these algorithms in time or implemented with the performance issue.

In Keras and Caffe, compatibility debt accounts for more than 10%; thus, it deserves attention too. Keras does not implement code that handles low-level operations such as tensor products, convolutions, and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library, such as Theano (which is not maintained anymore[43]), to do so; it serves as the *backend engine* of Keras[44]. Hence, Keras enjoys the convenience provided by backend projects at the cost of the maintenance of dependencies.

The majority of the SATD comments in deep learning framework are design debt (24.07% - 65.27%), requirement debt (7.09% - 31.48%) and algorithm debt (5.62% - 20.67%). In some projects, compatibility debt accounts for more than 10%.

5 DISCUSSION

In this section, we present our discussion on the comparison between the technical debt in deep learning frameworks with that in non-deep learning projects, implications for developers and researchers, and the threats to validity.

5.1 Comparison with Non-Deep Learning Projects

Our research investigates technical debt in deep learning frameworks by quantifying the number of technical debt instances at different granularities, classifying identified technical debt into different technical debt categories. However, prior researches have investigated the SATD in non-deep learning projects from these perspectives as well [19], [18], [17]. In this section, we inspect the similarities and differences of the technical debt between the deep learning frameworks and non-deep learning projects.

Table 4 presents the comparison results between our study and former researches. On average, 2.93% of all the comments are SATD comments in deep learning frameworks. This indicates that similar

Table 4: Comparison between our findings and prior studies

Topic	Prior Study	Our Study
Proportion of SATD comments	1.86% of all the comments. [19].	2.93% of all the comments.
Proportion of the files that contain one or more SATD instances	10.4% on average. [23]	17.27% on average.
Proportion of the classes that contain one or more SATD instances	1.6% on average. [23]	7.01% on average.
Proportion of the methods that contain one or more SATD instances	0.98% on average. [23]	1.82% on average.
SATD category	design debt, defect debt, requirement debt, test debt, and documentation debt [18].	design debt, defect debt, requirement debt, test debt, documentation debt, algorithm debt and compatibility debt.
SATD category distribution	the majority of SATD are design debt(42% - 84%), followed by requirement debt(5% - 45%.). Maldonado and Shihab [18]	the majority of the SATD in deep learning frameworks are design debt (24.07% - 65.27%), followed by requirement debt (7.09% - 31.48%) and algorithm debt (5.62%–20.67%).

to the development process in non-deep learning projects, developers involved in deep learning frameworks write down comments to record their temporary implementation of the sub-optimal solution as well. However, the proportions of the comments that indicate technical debt among all the comments in deep learning frameworks are 57.5% larger than those in non-deep learning projects, and at any level of granularity, the proportions of files/ classes/ methods in deep learning frameworks that contain one or more SATD instances are 65.4%, 338.1%, and 85.7% larger than those in non-deep learning frameworks. This indicates that the SATD is more common in deep learning frameworks compared to non-deep learning projects.

Moreover, compared with developers in non-deep learning projects, developers in deep learning frameworks are confronted with new challenges. As is indicated in Section 4.3, besides the technical debt that is discovered in non-deep learning projects, there is algorithm debt and compatibility debt in deep learning frameworks as

[43]https://www.quora.com/Is-Theano-deep-learning-library-dying
[44]https://keras.io/

well. Furthermore, as for the categories of technical debt that are common to non-deep learning projects, they also have additional deep learning characteristics. For example, many deep learning algorithms are probability and statistics-based algorithms. Such algorithms are sensitive to the input data. This leads to defects in deep learning projects that are hard to reproduce. Developers write down the defects that are postponed repairing and leave such defects as defect debt in deep learning frameworks. Besides, deep learning algorithms cover a wide range of knowledge. And it is difficult for a developer to master a wide range of knowledge and have a good understanding of the project. Therefore, many requirements remain uncomplished, and many developers expect other developers or experts to professionally write the documentation of the code. Moreover, deep learning algorithms are still rapidly advancing. Developers have to sacrifice the quality of projects to speed up the development time in order to introduce these new algorithms to win in the fierce competition.

5.2 Implications

For application developers, our research finds that there are many instances of technical debt in deep learning frameworks. We provide the following suggestions for application developers:

(1) We suggest application developers be careful about the frameworks with more defect debt, such as CNTK. The defect debt can lead to the unstable data dependencies in applications.
(2) If the deep learning application needs to use the out-of-box implementation of the cutting-edge deep learning algorithms provided by deep learning frameworks, we suggest the application developers not to use the frameworks with more requirement debt, such as DL4J, where many algorithms remained un-implemented.
(3) When application developers employ Keras in their projects, they should be careful about the compatibility of the Keras on the dependencies. This is because the compatibility debt leads to more changes on related files, which is prone to introduce defects.
(4) If the deep learning application requires a high performance, we suggest application developers not to employ the frameworks with more algorithm debt, such as Keras and DL4J.

For framework developers, our research finds that there is technical debt in deep learning frameworks, and many of them remained un-removed. Due to the negative impact of technical debt, **we suggest framework developers resolve technical debt in time**. We provide the possible solutions to remove or minimize the impact of technical debt:

(1) For design debt, framework developers can refactor the code, detect duplicated code, and ask expert developers to re-implement the poor implementation.
(2) For documentation debt, framework developers can (1) provide qualified documentation in time, (2) ask the experts how to express the derivation of the algorithm, or (3) provide a reference to a paper.
(3) For defect debt, framework developers can (1) resolve the defects and (2) post the unsolved defects in the issue tracking system to inform other developers.
(4) For requirement debt, framework developers can (1) implement the requirements in time and (2) notify the incompleteness of the task in release note.
(5) For test debt, framework developers can (1) add sufficient test cases, and (2) remove dead tests.
(6) For compatibility debt, Sculley et al. [25] suggest developers re-implement specific functions within the broader system architecture, which makes the framework not bind themselves tightly with other dependencies.
(7) For algorithm debt, since it is difficult for developers to both master the project and the complex algorithms, we suggest developers can (1) collaborate with one another, or (2) ask experts for help.

For researchers in software engineering, our findings illustrate that there are compatibility debt and algorithm debt in deep learning framework projects. However, compatibility debts might be prevalent in projects that heavily rely on upstream libraries. With the growth of open-source libraries, more developers would employ open-source libraries in their projects. This practice would lead to the prevalence of compatibility debt in the future. Besides, algorithm debt would also be widespread in performance-critical and algorithm intensive projects, e.g., game engine. We suggest researchers could investigate the generality of the algorithm debt and compatibility debt.

Furthermore, our findings illustrate the distribution of different types of technical debt in deep learning framework projects. However, it is still unclear how the distribution evolves along the development process. By doing so, we could better understand the trade-off performed by developers. Therefore, we encourage researchers to investigate how framework developers deal with different categories of technical debt in the future, e.g., which categories of technical debt are removed the most or the fastest.

For researchers in deep learning, we suggest deep learning researchers try to test their algorithms based on more deep learning frameworks, rather than only based on the interfaces provided by one deep learning framework.

5.3 Limitation

To identify technical debt in a project, we use source code comments that describe part of the source code containing technical debt. One threat of using source code comments is the consistency of changes between the comments and the code, i.e., in some cases the comment may change but not the code and vice versa. However, previous work showed that between 72-91% of the code and comment changes are consistent, i.e., code and comments co-change together [23].

To classify the detected source code comments into different categories, we heavily depended on manual process. Like any human activity, our manual classification is subject to personal bias and subjectivity. To reduce personal bias in manual classification of code comments, as we indicate in Section 4.3, the first author selects another 2,000 comments from the 234,260 comments and invite an independent Ph.D. student, who is not an author of this paper, to manually classify them. Then, a high level of agreement between the classification given by the Ph.D. student and the first author is reported with Cohen's kappa coefficient of +0.79. This gives us high confidence in the dataset used in our paper.

Threats to external validity concern the generalization of our findings. Our study is conducted on seven large open source deep learning frameworks. That said, our findings may not be generalized

to other open source or commercial projects. In the future, we will analysis technical debt in other systems.

6 CONCLUSION

In this paper, we investigate the technical debt in deep learning frameworks. We find that technical debt is common in deep learning frameworks, which indicates the use of deep learning frameworks is not free. TensorFlow has 3,775 technical debt, which is the most among the studied frameworks. The majority of the technical debt is design debt (24.07% - 65.27%), followed by requirement debt (7.09% - 31.48%) and algorithm debt (5.62%–20.67%). This suggests that the rapid advancement of the cutting-edge deep learning algorithms may have caused framework developers to not have implemented these algorithms in time or implemented them with a number of issues. There are defect debt, documentation debt, test debt, and compatibility debt as well. The majority of technical debt is design debt (24.07% - 65.27%), followed by requirement debt (7.09% - 31.48%) and algorithm debt (5.62%–20.67%). In some projects, compatibility debt accounts for more than 10%, which cannot be ignored. Based on our findings, we provide suggestions for application developers and the possible methods to resolve the technical debt for framework developers.

In the future, we will investigate the generality of the algorithm debt in performance-critical and algorithm intensive projects, and the generality of the compatibility debt in projects that heavily relies on upstream libraries.

ACKNOWLEDGMENTS

This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904), NSFC Program (No. 61972339), and the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021).

REFERENCES

[1] Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo O Spínola. 2014. Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 1–7.

[2] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 315–326.

[3] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 47–52.

[4] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.

[5] Joseph L Fleiss and Jacob Cohen. 1973. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement* 33, 3 (1973), 613–619.

[6] Francesca Arcelli Fontana, Vincenzo Ferme, and Stefano Spinelli. 2012. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 15–22.

[7] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[8] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 78–87.

[9] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 485–495.

[10] Yasutaka Kamei, Everton da S Maldonado, Emad Shihab, and Naoyasu Ubayashi. 2016. Using Analytics to Quantify Interest of Self-Admitted Technical Debt.. In *QuASoQ/TDA@ APSEC*. 68–71.

[11] Noela Jemutai Kipyegen and William P K Korir. 2013. Importance of Software Documentation. *International Journal of Computer Science Issues* 10, 5 (2013), 6.

[12] Tim Klinger, Peri Tarr, Patrick Wagstrom, and Clay Williams. 2011. An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on managing technical debt*. ACM, 35–38.

[13] Rainer Koschke. 2007. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[14] Philippe Kruchten, Robert L Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes* 38, 5 (2013), 51–54.

[15] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[16] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: What software practitioners have to say about technical debt. *IEEE software* 29, 6 (2012), 22–27.

[17] Everton Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An Empirical Study On the Removal of Self-Admitted Technical Debt. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE.

[18] Everton Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *Proceedings of the 7th IEEE International Workshop on Managing Technical Debt (MTD'15)*. 9–15.

[19] Everton Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[20] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[21] Roman Novak, Yasaman Bahri, Dan Abolafia, Jeffrey Pennington, and Jascha Sohl-dickstein. 2018. Sensitivity and Generalization in Neural Networks: an Empirical Study. https://openreview.net/pdf?id=HJC2SzZCW

[22] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[23] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 91–100.

[24] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[25] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.

[26] Carolyn Seaman and Yuepu Guo. 2011. Measuring and monitoring technical debt. In *Advances in Computers*. Vol. 82. Elsevier, 25–46.

[27] Securities and E. Commission. [n. d.]. SEC Charges Knight CapitalWith Violations ofMarket Access Rule. https://www.sec.gov/news/press-release/2013-222. 2013.

[28] Hai Shu and Hongtu Zhu. 2019. Sensitivity Analysis of Deep Neural Networks. (01 2019).

[29] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[30] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

[31] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 179–188.

[32] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the Impact of Self-admitted Technical Debt on Software Quality. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. 11.

[33] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 17–23.

[34] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt (MTD '11)*. ACM, New York, NY, USA, 17–23. https://doi.org/10.1145/1985362.1985366

[35] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. 2013. A Case Study on Effectively Identifying Technical Debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE '13)*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/2460999.2461005

[36] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. *International Symposium on Software Testing and Analysis* (2018).