DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions

Danning Xie
Purdue University
West Lafayette, IN, USA
xie342@purdue.edu

Yitong Li
University of Waterloo
Waterloo, ON, Canada
ytnli95@gmail.com

Mijung Kim*
Ulsan National Institute of Science and Technology
Ulsan, South Korea
mijungk@unist.ac.kr

Hung Viet Pham
University of Waterloo
Waterloo, ON, Canada
hvpham@uwaterloo.ca

Lin Tan†
Purdue University
West Lafayette, IN, USA
lintan@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, IN, USA
xyzhang@cs.purdue.edu

Michael W. Godfrey
University of Waterloo
Waterloo, ON, Canada
migod@uwaterloo.ca

ABSTRACT

Input constraints are useful for many software development tasks. For example, input constraints of a function enable the generation of valid inputs, i.e., inputs that follow these constraints, to test the function deeper. API functions of deep learning (DL) libraries have DL-specific input constraints, which are described informally in the free-form API documentation. Existing constraint-extraction techniques are ineffective for extracting DL-specific input constraints.

To fill this gap, we design and implement a new technique—*DocTer*—to analyze API documentation to extract DL-specific input constraints for DL API functions. DocTer features a novel algorithm that automatically constructs rules to extract API parameter constraints from syntactic patterns in the form of dependency parse trees of API descriptions. These rules are then applied to a large volume of API documents in popular DL libraries to extract their input parameter constraints. To demonstrate the effectiveness of the extracted constraints, DocTer uses the constraints to enable the automatic generation of valid and invalid inputs to test DL API functions.

Our evaluation on three popular DL libraries (TensorFlow, PyTorch, and MXNet) shows that DocTer's precision in extracting input constraints is 85.4%. DocTer detects 94 bugs from 174 API functions, including one previously unknown **security vulnerability** that is now documented in the CVE database, while a baseline technique without input constraints detects only 59 bugs. Most (63) of the 94 bugs are previously unknown, 54 of which have been fixed or confirmed by developers after we report them. In addition, DocTer detects 43 inconsistencies in documents, 39 of which are fixed or confirmed.

*The work was completed when Mijung Kim was at Purdue University
†Corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534220

CCS CONCEPTS

• **Software and its engineering** → **Software reliability**; **Software testing and debugging**.

KEYWORDS

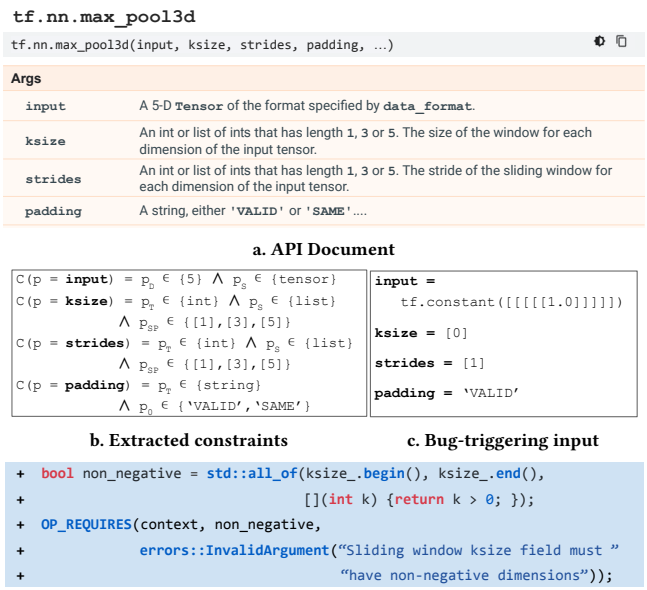
text analytics, testing, test generation, deep learning

ACM Reference Format:
Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions . In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534220

1 INTRODUCTION

Input constraints are useful for various software development tasks [16, 21, 32, 52, 56]. For example, input constraints of a function enable the generation of valid inputs, i.e., inputs that follow these constraints, to test the function deeper. API functions of DL libraries expect their input arguments to follow constraints, many of which are DL-specific. For example, one parameter `input` of the PyTorch API function `torch.as_strided` has to be a tensor. A *tensor* is represented using an n-dimensional array, where *n* is a non-negative integer. Any input that cannot be interpreted as a tensor (e.g., a Python list) is invalid. Many such DL-specific input constraints are described informally in free-form API documentation. The availability of such DL API documentation presents a great opportunity to automatically extract DL-specific constraints for better testing and other software development tasks.

Specifically, DL libraries' API functions require two types of constraints for their input arguments: (1) data structures and (2) properties of these data structures. First, DL libraries often require their input arguments to be specific *data structures* such as lists, tuples, and tensors to perform numerical computations. For example, `input` of the PyTorch API function `torch.as_strided` has to be a tensor as dictated by its API document. Any input that cannot be interpreted as a tensor (e.g., a Python list) is rejected by the function's input validity check. Such invalid inputs exercise only the input validity checking code, failing to test the core functionality of



a. API Document

b. Extracted constraints

c. Bug-triggering input

d. Bug fix in `pooling_ops_3d.cc`

Figure 1: TensorFlow document helps our tool detect a bug that was fixed after we reported it to TensorFlow developers.

the API function. To test `as_strided`'s core functionality, a testing technique needs to generate a tensor object for the `input` parameter.

Second, API functions of DL libraries require their arguments to satisfy specific *properties* of data structures. Generating a correct data structure with incorrect properties often fails the input validity checking of the DL API functions. They often require two common properties of a data structure—*dtype* and *shape*. Property *dtype* specifies the data type of the data structure (e.g., `int32`, `float64`, and `String`). In Fig. 1a, the *dtype* of the parameter `padding` should be `String`. Property *shape* specifies the length of each dimension of the data structure. For example, a *shape* of 3×4 matrix is a 2-dimensional tensor with the first dimension of 3 elements and the second dimension of 4 elements. As another example, Fig. 1a shows the document for TensorFlow API `tf.nn.max_pool3d`, which indicates that the parameter `input` should be a tensor of 5 dimensions, with the size of each dimension unspecified. Similarly, any inputs that violate these *dtype* or *shape* requirements are rejected, failing to test the core functionality of the API function.

While existing techniques can extract constraints from code or software text (e.g., comments and documents), they are insufficient for extracting DL-specific constraints. Specifically, while Pytype [7] infers data types from Python code, it cannot precisely infer types for DL libraries because it cannot analyze across Python and C++ code. In addition, it cannot extract numerical constraints such as *shape* and *range*. Existing techniques that derive constraints from software text extract different types of constraints that are not DL-specific, such as exceptions [16, 56], command-line options and file formats [52], locking [44], call-relations [31, 44], interrupts [45], nullness [46, 60] and inheritance relations [60]. Although some [16, 31, 56, 60] can extract constraints related to valid ranges, those are only a small portion of DL-specific constraints (Section 4.1). Techniques such as C2S [56] require pairs of Javadoc comments and formal JML [1] constraints as input. For DL API functions, such formal constraints are unavailable.

1.1 Our Approach

To fill this gap, we design and implement a new technique—*DocTer*—to analyze API documentation to extract DL-specific input constraints. DocTer features a novel method to automatically derive *constraint extraction rules* from a small set of manually annotated API documents with precise constraint information. These rules can predict API parameter constraints based on syntactic patterns in the form of dependency parse tree in documents. They are then applied to the full sets of API documents of popular DL libraries to extract constraints.

To demonstrate the effectiveness of these extracted constraints, DocTer uses them to guide and improve an important task — generating test cases automatically to test DL API functions. Testing API functions of DL libraries (e.g., TensorFlow [14] and PyTorch [38]) is crucial because these libraries are widely used and contain software bugs [26, 27, 41, 57, 58], which hurt not only the development but also the accuracy and speed of the DL models.

Yet, generating test cases for DL libraries' API functions is challenging. If a test-generation tool is (1) unaware of DL-specific constraints or (2) incapable of using these constraints to generate diverse inputs, it is practically impossible to generate valid inputs to reach deeper states and test the core functionality of DL API functions. Existing test-generation tools [3, 5, 9, 19, 37, 43] such as AFL [3] and libFuzzer [5] have no knowledge of such input constraints, thus are very limited in testing DL API functions. DocTer addresses these challenges by using the following techniques:

(1) DL-specific constraint extraction: Since API documents are written informally in a natural language, manually extracting constraints from a large number of API documents (e.g., TensorFlow v2.1.0 has 2,334 pages of API documents and 854,900 words) is inefficient and tedious. In addition, since these documents are constantly evolving, it is undesirable and error-prone to manually analyze them each time the documents are updated which can be as frequent as every commit. To address these challenges, we develop a novel method that can automatically derive a set of rules that predict parameter constraints from parse tree patterns of API description. Given a small set of API function descriptions and the corresponding constraint annotations, DocTer identifies a set of rules as an optimal mapping that can minimize prediction errors and achieve the maximum coverage of constraints. By applying these constructed rules to a much larger set of real-world documents, DocTer can automatically extract DL-specific constraints for API functions of the most widely used libraries.

(2) DL-specific input generation: After extracting DL-specific input constraints (e.g., Fig. 1b), DocTer uses these constraints to guide test generation to produce valid inputs (e.g, Fig. 1c), invalid inputs, and boundary inputs (such as `-MaxInt`, `0`, and `MaxInt` for the constraint of *dtype* of `int`). DocTer evaluates valid inputs by checking if the API runs successfully without failures, e.g., crashes. If a failure occurs with a valid input, the generated test has manifested a bug in the implementation of the API's core functionality.

Fig. 1 shows a previously unknown bug detected by DocTer in TensorFlow along with its patch that the TensorFlow developers

committed after we reported the bug. The API document in Fig. 1a indicates that the shape of `input` is 5-D, and `ksize` is an integer or a list of 1, 3, or 5 integers. DocTer automatically extracts the constraints in Fig. 1b and generates the bug-triggering input in Fig. 1c. Detailed constraint formats are explained in Section 2.2. DocTer generates a valid input. Specifically, parameter `input` is a five-dimensional (5-D) tensor as a constant (`tf.constant`), where the five pairs of square brackets denote a five-dimensional tensor. Parameter `ksize` is a list of length 1, whose element is a zero (i.e., `[0]`), parameter `strides` is `[1]`, and parameter `padding` is `"VALID"`.

This bug is only triggered when the parameter `ksize` has a zero value. This zero value causes a division-by-zero fault, resulting in a floating point exception. To trigger this bug, the parameter `padding` must be either `"VALID"` or `"SAME"`. Otherwise the function's input validity checking would reject the input with an `InvalidArgumentError`. Therefore, it is practically impossible for techniques that randomly generate inputs to trigger this bug. After we reported this bug, the TensorFlow developers added the `non_negative` range validation for the parameter `ksize` (Fig. 1d).

In addition, DocTer generates invalid inputs that violate the constraints to detect crashes. Despite invalid inputs, DL API functions should not crash. Instead, they are expected to report an invalid input (e.g., by throwing an exception or printing an error message). This point is well confirmed by an API developer after we reported a crash bug detected by DocTer "*A segmentation fault is never OK and we should fix it with high priority*". Such invalid-input generation is impossible without the constraints.

(3) Documentation-bug detection: Since incorrect API documentation provides false information about APIs, which often misleads API users to introduce bugs in code [44], it is important to detect bugs in API documents as well. Different from prior work [44, 46] that detects inconsistencies between documents/comments and code, DocTer detects inconsistencies within documents. For example, in the document of `tf.keras.backend.moving_average_update`, the description for the parameter `value` is "*...with the same shape as variable,...*", but the parameter `variable` is not documented. This documentation bug of erroneous parameter dependencies has been fixed after we report it.

1.2 Contributions

In this paper, we make the following contributions:

- A novel rule construction technique that formulates the challenge as an optimization problem aiming to find the smallest set of rules that can make the largest number of correct extractions of parameter constraints. We also develop an approximate solution to the problem based on sample space conditional probability computation.
- A document-analysis technique that extracts 16,035 constraints automatically from API documentation with the focus on four categories of input properties in DL APIs: *structure*, *dtype*, *shape*, and *valid values* for 2,415 API functions across the three widely-used DL libraries, TensorFlow [14], PyTorch [38], and MXNet [17]. The constraint extraction precision is 85.4%.
- An application of our extracted constraints to guide the generation of DL-specific inputs.
- A tool *DocTer* that combines the techniques above, and detects 94 bugs in 174 APIs from the three libraries, while a baseline that generates inputs without the knowledge of constraints detects 59 bugs only. Among the 94 bugs, 63 are previously unknown bugs, 54 of which have been fixed (49) or confirmed (5) by the developers after we report them. Notably, **one of the previously unknown bugs was added to the CVE vulnerability database** for TensorFlow after we reported it. In addition, DocTer detects 43 documentation bugs, 39 of which have been fixed (35) or confirmed (4) after we report them.

While our rule construction and constraint extraction techniques are general, the constructed rules and extracted constraints are domain-specific. We focus on testing DL libraries due to their importance and the lack of available constraint-extraction techniques for them. We leave the extension to other domains, e.g., classic machine learning libraries such as scikit-learn [39], as future work.

Availability: We share the tool DocTer, bug list, and data in [13].

2 APPROACH

2.1 Overview

Fig. 2 shows the overview of DocTer using an example of the TensorFlow API `tf.nn.atrous_conv2d`. DocTer consists of three phases. The *rule construction phase* (i.e., the green box in Fig. 2) takes a small portion of API documents with annotations to construct a set of rules that can extract concrete constraints from API documents. The rules are constructed by an optimization-based method. They are mappings from document dependency parse trees to the corresponding abstract parameter constraints in the form of assertions (e.g., on *dtype* and *shape*), which are called *Abstract Constraints* (ACs). In the *constraint extraction phase* (i.e, the orange box), the rules are applied to concrete API documents to derive concrete parameter constraints. To demonstrate the effectiveness of these extracted constraints, in the *testing phase* (the purple box), DocTer generates test inputs either conforming or violating the constraints (by the *input generator*), and executes the inputs to detect bugs (by the *test case evaluator*), in an iterative fashion.

A major challenge of constraint extraction is analyzing free-form API documentation written in the natural language [16, 46, 52]. We observe that developers have limited ways to express input constraints in natural language. However, these expressions are instantiated differently for different APIs and composed together in various ways, leading to complex overall syntactic structures that are difficult to translate to parameter constraints. We hence devise a novel method that works as follows. It first preprocesses/normalizes the documents to dependency parse trees and then breaks the trees into subtrees. With a small set of API documents and the corresponding manually annotated ACs, an algorithm is developed to identify the optimal mappings between subtrees and parameter constraints that can maximize the matching of the mapped constraints and the ground-truth annotations. These mappings are essentially our constraint extraction rules. DocTer applies these rules to extract constraints from API documents automatically. Apart from a fixed cost of annotating a small portion (e.g., 30%) of API parameters, our process is automatic and can be reapplied to future versions or another relevant library with little manual work.

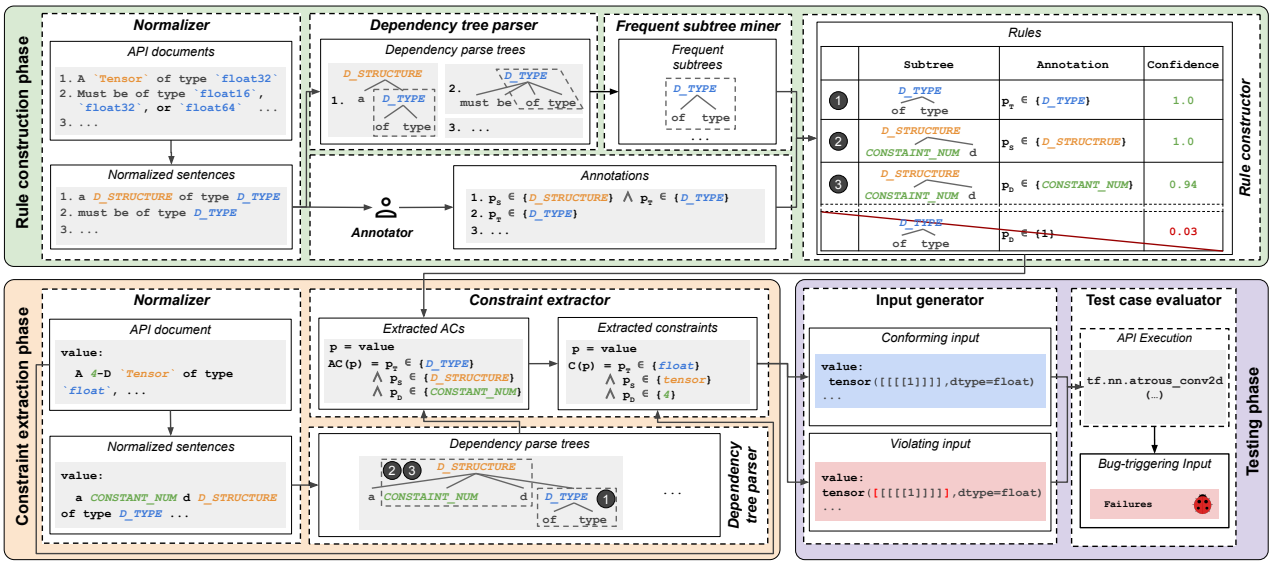


Figure 2: Overview of DocTer. p_T, p_S, and p_D are the abstract constraints representing the *data type*, *data structure*, and *number of dimensions* of parameter p, respectively. Detailed constraint formats are explained in Section 2.2.

Preprocessing: During both the rule construction and constraint extraction phases, DocTer performs two preprocessing steps: *normalization* and *dependency tree parsing* to convert free-form API descriptions to dependency parse trees (parse trees for short). (Fig. 2). The normalizer replaces keywords with abstractions, e.g., replacing data type keywords (e.g., `int32`, `float64`) with `D_TYPE`, structure type keywords (e.g., `Tensor`, `list`) with `D_STRUCTURE`, and integer constants with `CONSTANT_NUM`. This normalization improves the performance of the rule construction algorithm by suppressing instance differences. We use dependency tree parser [33] to convert the normalized sentences to parse trees.

An example: Our rule construction component identifies a rule ❶ (row one of the *Rules* table in Fig. 2) that maps a frequently occurring subtree pattern "*of type* `D_TYPE`" (e.g., appearing in both sentences 1 and 2 in Fig. 2) to an abstract type assertion (an AC) $p_T \in$ {`D_TYPE`}, which means that the valid *dtype* of parameter p should be one from the set {`D_TYPE`}, where `D_TYPE` is an abstraction of one or more *dtypes*, which, in this example, are `float16`, `float32`, and `float64`. In the annotated dataset, the conditional probability of the type assertion $p_T \in$ {`D_TYPE`}, given the subtree pattern is 1.0 and the pattern is the smallest with such predictive power. Thus, the rule constructor is able to create rule ❶.

The *constraint extractor* applies constructed rules to all the preprocessed API documents to automatically extract a set of constraints for each input parameter. For example, in the Tensorflow document for API `tf.nn.atrous_conv2d`, one of the parse trees parsed from the description for parameter `value` (e.g., Fig. 2) contains two frequent subtrees *"a* `CONSTANT_NUM` *d* `D_STRUCTURE`*"* and *"of type* `D_TYPE`*"*. These structures correspond to rules ❶, ❷, and ❸. DocTer applies these rules and obtains the extracted ACs for the parameter in Fig. 2 (the middle of the orange box), e.g., p = *value*; $AC(p) = p_T \in$ {`D_TYPE`} $\wedge\, p_S \in$ {`D_STRUCTURE`} $\wedge\, p_D \in$ {`CONSTANT_NUM`} , where p_T, p_S and p_D represent the *data type*, *data structure* and *number of dimensions* of parameter p, respectively. DocTer further instantiates the abstract symbols (`D_TYPE`, `D_STRUCTURE`, and `CONSTANT_NUM`) with the corresponding value and types (i.e., `float`, `Tensor`, and 4) from the original sentence to convert the ACs to concrete constraints. We now discuss each individual step.

2.2 Preprocessing

The first step of DocTer is to collect the natural language API documents. They are at high volume. For example, there are 2,334 pages of API documents and 854,900 words in TensorFlow v2.1.0. It is hence a daunting and tedious task for developers to manually examine such a large set of API documents to identify constraints.

API document collection and tokenization: After collecting the API documents (in the form of HTML pages from DL libraries' websites), DocTer parses these files to obtain API signatures and parameter descriptions with an HTML parsing tool [2]. Since sentence is a natural unit of organizing constraints, DocTer further splits the description into sentences with a sentence segmentator [15].

Normalization: The tokenized sentences are normalized. While developers may have a small number of patterns expressing parameter constraints, these patterns have diverse instantiations according to the concrete data types and parameters involved. Normalization abstracts away these instantiation differences.

Specifically, DocTer normalizes keywords such as (1) data types (e.g., `int32`) and (2) data structures (e.g., `tensor`) as `D_TYPE` and `D_STRUCTURE`, respectively. To get the list of keywords for data types, we collect a list of supported data types from each library [10–12]. We then expand such a list with informal variations (e.g., "integer", and "ints") and missing common types (e.g., `String`) to match the format of API documents. In total, we use 84, 74, and 53 type keywords for TensorFlow, PyTorch, and MXNet, respectively. The full list of keywords can be found in [13].

DocTer also normalizes constants such as (3) integer, (4) float, (5) boolean values as CONSTANT_NUM, CONSTANT_FLOAT, and CONSTANT_BOOL. It also replaces (6) relational expressions (e.g., "≥ 1") with REXPR and replaces (7) parameter names with PARAM.

The content that is (8) quoted often refers to enumerate values, so DocTer replaces such content with ENUM. For example, *"'NWC' and 'NCW' are supported."* is normalized to *"*ENUM *are supported"*. The shape of a parameter is often put within (9) a pair of square brackets or parentheses, DocTer replaces such content with SHAPE. For example, *"A Tensor of shape [num_classes,dim]"* is normalized to *"A* D_STRUCTURE *of shape* SHAPE*"*. Finally, consecutive abstract annotations of the same type are replaced with just one. For example, the three type keywords in *"Must be of type 'float16', 'float32', or 'float64'."* (Fig. 2) are replaced by a single D_TYPE, resulting in a normalized sentence *"must be of type* D_TYPE*"*.

Dependency tree parsing: Once the sentences are normalized, they are fed to the dependency tree parser [33], which conducts POS-Tagging and builds tree structure relationships (i.e., dependency parse trees) between words of a sentence based on the grammatical structure. For example, in the sentence *"a* D_STRUCTURE *of type* D_TYPE*"* from Fig. 2, the words "D_STRUCTURE", *"type"*, and "D_TYPE" are first tagged as NN (noun). Then the parser conducts dependency parsing and generates the dependency parse tree as shown in the figure where D_STRUCTURE is the root, and D_TYPE is the *nominal modifier* [4] of the root.

Annotating a subset of API descriptions with ACs: To support rule construction, we randomly pick a small set of the parameters (30%) and manually annotate them with their ACs. To minimize possible biases, the process involves three co-authors. Two authors independently annotate with 98.2% agreement. All disagreements are resolved with a third author to reach a consensus.

Abstract Constraints (AC): ACs are abstract constraints/assertions. These assertions are not on concrete *dtype* or *shape* but rather abstract ones. An AC for a parameter p is denoted in the form of $p_t \in \{T_1, T_2, \dots\}$ where t is the category of AC, and T_1 and T_2 are the possible abstract values. For example, $p_T \in \{\text{D\_TYPE}\}$ means that p is of D_TYPE. Specifically, the annotations of parameter p are designed as follows:

- p_T denotes the *data type* of an abstract constraint (AC) of p. .
- p_S denotes the *data structure* of an AC of p.
- $p_{SP} \in \mathbf{N}^D$ where $D \in \mathbf{N}$ denotes the *shape* AC of p, where D represents the number of dimensions of p.
- $p_D \in \mathbf{N}$ denotes the *number of dimensions* of an AC of p. Therefore, $p_D = p_{SP}.length$ if p is a tensor or $p_D = 0$ if p is a scalar.
- p_i denotes an element in parameter p if p is a tensor, where $i = 1, 2, \dots, Prod(p_{SP})$. When p is a scalar, its value is p_0.

An AC can be instantiated to different concrete constraints. Table 1 provides examples of ACs (first column as part of the rules) and their instantiations (last column) for several APIs.

AC annotation categories: We focus on annotating four categories of ACs (i.e., *structure*, *dtype*, *shape*, and *valid value*) because they represent the most common (93.6%) properties of input parameters of API functions in major DL libraries. The four categories are:

- *structure*: the type of data structure that stores a collection of values for the input parameter, such as list, tuple, and n-dimensional array (i.e., tensor).
- *dtype*: the data type, such as int, float, boolean, and String, of the parameter or the elements of *structure*.
- *shape*: the shape or number of dimensions of the parameter. For example, in row 2 of Table 1, weights is of shape [num_classes, dim] (i.e., it is a 2-D array where the sizes of its first and second dimensions are num_classes and dim, respectively).
- *valid value*: a set of valid values (e.g., parameter padding can only be either "VALID" or "SAME") or the valid range of a numerical parameter (e.g., a float between 0 and 1).

We make reasonable assumptions when annotating API descriptions. For example, a parameter is assumed to be a 0-dimensional non-negative integer if the document states it is a *"number of ..."*. *The assumptions are in the supplementary material [13].*

2.3 Rule construction

Although API descriptions are in a natural language, these descriptions often share a small number of syntactic patterns. For example, a constraint of *dtype* assertion is mostly described by two syntactic patterns: *"must be one of the following types ..."* and *"a tensor of type ..."* in TensorFlow. Our idea is hence to identify such patterns in API descriptions and project them to the corresponding parameter constraints. We call such projections the *constraint extraction rules*.

Automatically deriving such rules is challenging. The first challenge is that a syntactic pattern may have different instantiations in various API descriptions, depending on the variables and types. For example, the aforementioned pattern *"A 'Tensor' of type ..."* is instantiated to *"A 'Tensor' of type 'string'"* and *"A 'Tensor' of type 'int32'"* in two respective parameters contents and crop_window in API tf.io.decode_and_crop_jpeg. Our normalization step substantially mitigates this problem. The second challenge is that such patterns are often convoluted in the overall syntactic structure of an API description. For example, consider the description of parameter value as shown in Fig. 2. The normalized sentence *"a* CONSTANT_NUM *d* D_STRUCTURE *of type* D_TYPE.*"* is composed of two syntactic patterns *"*CONSTANT_NUM *d* D_STRUCTURE*"* and *"of type* D_TYPE*"*. Third, these patterns may have arbitrary sizes.

An optimization problem: We propose a novel method to automatically derive the extraction rules from a small set of APIs with their ACs manually annotated. We formulate it as an optimization problem. Specifically, given an API f, its normalized natural language description is denoted as D_f, its ACs are denoted as A_f. We use $trees(D_f)$ to denote all the subtrees of the parse tree of D_f. For example, Fig. 2 gives a 3-layer parse tree of the normalized sentence *"a* D_STRUCTURE *of type* D_TYPE*"*, which has subtrees *"of type* D_TYPE*"* and *"a* D_STRUCTURE *type"*. Such subtrees consider both parent-child (direct) connections and ancestor-descendant (indirect) connections. We use *Tree* and *AC* to denote the domains of subtrees and ACs, respectively. Our goal is hence to derive a mapping $R : Tree \rightarrow AC$. The mapping should satisfy the following optimization objectives. First, the tree patterns can be used to precisely predict the corresponding ACs. If we consider the tree patterns and the ACs form a distribution, the conditional probability of an AC given the condition of its tree pattern shall be high. Second, all

ACs can be predicted by these patterns, i.e., our mappings should be comprehensive. Third, the number of the mappings from a tree pattern to an AC in R is minimum. The objective is needed otherwise a simple solution would be to include all tree patterns in descriptions. Fourth, the size of each tree pattern is minimum. It is very likely that a tree pattern and its sub-patterns both can predict a constraint. In such cases, we prefer the smallest one, which provides the maximum generalization.

Formally, the process to find the set of rules, that is, the optimal mappings R, is the following.

$$\arg\min_{R} \mathbb{E}_{(D_f,A_f)\sim\mathcal{N}} \left[\begin{array}{l} \dfrac{\sum\limits_{a\in A_f, t\in trees(D_f), R(t)=a} 1-P(a|t)}{|A_f|} + \\ |\{a \in A_f \,|\, \nexists t \in trees(D_f)\ s.t.\ R(t) = a\}| + \\ |R| + \sum\limits_{a\in A_f, t\in trees(D_f), R(t)=a} |t| \end{array} \right] \tag{1}$$

Here, $\mathcal{N}$ denotes the distribution of API description and the corresponding ground-truth ACs. The above formula means that we are looking for an R that can minimize the expected objective function value for all samples $(D_f, A_f) \sim \mathcal{N}$. The objective function is the sum of four terms. The first one is the average conditional probabilities for all the ACs in A_f. Intuitively, it means that for each AC a in an API, our mapping R should associate a tree pattern t with a such that the conditional probability $P(a|t)$ is maximum. The second term means that the number of ACs in A_f for which R does not have a mapping is minimum. This is to maximize the coverage of our rules. The third term is to minimize the size of R. The last term is to minimize the size of each tree pattern in R.

Approximate solution: Solving the above optimization problem is difficult because it is discrete. Its complexity is NP. This is not a typical learning problem as it does not aim to learn a distribution but rather to construct a minimum and yet complete set of rules. In addition, the amount of data available for training is relatively small compared to other domains that have successful applications of deep learning models. We hence devise an approximate solution. The first term can be approximated by computing the sample space conditional probabilities and then including the top associations in R. The sample space conditional probabilities are:

$$\overline{P}(a|t) = \frac{|\{f \mid a \in A_f \wedge t \in trees(D_f)\}|}{|\{f \mid t \in trees(D_f)\}|} \tag{2}$$

Intuitively, it is the number of co-occurrences of a tree pattern and an AC divided by the number of occurrences of the tree pattern. We further observe that if a tree pattern t is rare, it is usually not related to parameters. As such, we can focus on the frequent subtrees. We use frequent subtree mining [55] to efficiently discover the most frequent subtrees. We filter out the rare tree patterns with threshold `min_support`, i.e., any tree patterns that occur less than or equal to `min_support` times are discarded. The second and third terms are approximated by selecting only the associations (of a and t) with a large sample space conditional probability. Specifically, DocTer includes in R associations (of a and t) with $\overline{P}(a|t)$ greater than or equal to `min_confidence` (the selection of and `min_support` and `min_confidence` is a trade-off between precision and recall and will be discussed in Section 3). An alternative to approximate the second term is to use a greedy algorithm to include additionally needed tree patterns to achieve (full) coverage of ACs. However, this often contradicts the third term. Empirically (see Section 4.1), we find that including the top associations provides a good balance. To approximate the fourth term, which minimizes the tree patterns, we keep only the smallest tree pattern when multiple patterns can be used to predict an AC.

Algorithm 1 Rule Construction

```
1:  function RuleConstruction(Sample, min_support, min_confidence)
2:      parser ← DependencyTreeParser()
3:      D_f, FreqD_f ← ∅
4:      R ← ∅
5:      foreach sentence ∈ Sample do
6:          dependencyParseTree ← parser.parse(sentence)
7:          D_f.add(dependencyParseTree)
8:      end foreach
9:      FreqD_f ← getFreqSubtree(D_f, min_support, MAX_SIZE)
10:     foreach t ∈ FreqD_f do
11:         foreach a ∈ selectAC(Sample, t, min_confidence) do
12:             R.add(t → a)
13:         end foreach
14:     end foreach
15:     return R
```

Algorithm 1 formalizes the process of finding the approximated solution. For each sentence in the annotated data (`Sample`), the dependency tree parser parses the sentence and generates the parse tree, which is added to the set D_f (lines 5–7). Then, we select the set of frequent tree patterns $FreqD_f$ whose frequencies are at least `min_support` using frequent subtree mining (getFreqSubtree) (line 9). This process keeps only tree patterns whose size is smaller than or equal to `MAX_SIZE`. For each frequent tree pattern t, selectAC selects a set of ACs (a) with probabilities $\overline{P}(a|t)$ greater than or equal to `min_confidence`. Then, each association of a and t is added to the set R (lines 10–14).

Table 1 shows examples of the automatically discovered rules by DocTer (col. "Extraction rules") and examples of matched sentences (col. "API sentences"). For example, rule ❸ in Table 1 is used to extract the enumerated value (e.g., *valid value*) of parameter p, which is associated with the AC $p_0 \in ENUM$. In rule ❹, the pattern *"number of"* implies parameter `p` should be a `0`-dimensional non-negative integer.

2.4 Constraint extraction

Given an API description, the constraint extractor matches the tree patterns in the rules with the parse tree of the description. Matches are then projected to the corresponding ACs, which are further instantiated in the context of the description to derive the concrete constraints. For example, in Fig. 2, the constraint extractor finds rules ❶, ❷, and ❸ match the two subtrees "a `CONSTANT_NUM` d `D_STRUCTURE`" and "of type `D_TYPE`" in the parse tree of the `value` description. DocTer then assigns the three relevant ACs to the parameter `value`. DocTer then instantiates the ACs with the concrete data types, structure types, and constants to generate the final constraints. In row 2 of Table. 1, the annotation `SHAPE` is instantiated

Table 1: Rule examples and the extracted constraints from TensorFlow, PyTorch, and MXNet

	Extraction rules		API sentences	Normalized sentences	Extracted constraints
1	type same PARAM as	AC(p) = p_T ∈ {&PARAM}	y: Must have the same type as `x`.	y: must have the same type as PARAM	p = y C(p) = p_T ∈ {&x}
2	D_STRUCTURE SHAPE of shape	AC(p) = p_S ∈ {D_STRUCTURE} ∧ p_{SP} ∈ {SHAPE}	weights: A `Tensor` of shape [num_classes,dim]	weights: a D_STRUCTURE of shape SHAPE	p = weights C(p) = p_S ∈ {tensor} ∧ p_{SP} ∈ {[&num_classes,dim]}
3	ENUM either	AC(p) = p_0 ∈ {ENUM}	padding: A string, either ``'VALID'`` or ``'SAME'``.	padding: a D_TYPE, either ENUM	p = padding C(p) = p_T ∈ {string} ∧ p_0 ∈ {'VALID','SAME'}
4	number of	AC(p) = p_T ∈ {int} ∧ p_D ∈ {0} ∧ p_i ∈ {[0,∞]}	num_group: Number of group partitions.	num_group: number of group partitions	p = num_group: C(p) = p_T ∈ {int} ∧ p_D ∈ {0} ∧ p_i ∈ {[0,∞]}

based on the original text, i.e., *"[num_classes, dim]"*. Row 3 in Table. 1 shows an example rule of *valid value* constraints. DocTer detects the pattern *"either ENUM"* and uses it to extract the *valid value* constraint in the last column.

Constraint dependencies graphs: The description of one parameter often refers to the *dtype* or value of another parameter from the same API function. In such cases, DocTer extracts constraints that involve *dependencies* among input parameters. These dependencies are useful not only for generating valid inputs but also for determining the parameters' generation order. The automatically constructed rules can detect *dtype* dependencies. For example, row 1 in the Table. 1) shows the pattern *"must have the same type as ..."* which indicates a type dependency. In this example, the operator '*&*' is to acquire the *dtype* of a parameter. An example of *shape* dependency is shown in row 2 of Table 1. Specifically, parameter `weights` should have shape `[num_classes,dim]` where the size of the first dimension is the value of another parameter `num_classes` while `dim` is a non-negative integer.

These dependencies are denoted in a graph with each edge representing a constraint dependency. During input generation, the graph is traversed in a topological order to ensure dependencies are properly considered. The graph construction is straightforward and hence elided.

2.5 Testing phase

To demonstrate the effectiveness of the extracted constraints, for each API function, DocTer iteratively generates an input i.e., values of the API function's parameters, and evaluates that input to detect crashes. By either following or violating the *extracted* constraints, the input generator generates *conforming inputs* (CIs) or *violating inputs* (VIs), respectively. The conforming inputs are designed to test the core functionality of the API function while the violating inputs aim to test the API function's input validity checking code. In both cases, DocTer reports bug-triggering inputs that cause serious crashes (e.g., segmentation fault). DocTer tests each API function with `maxIter` number of inputs, and the ratio of inputs allocated to each mode (CI or VI) is determined by the ratio `conform_ratio`.

Input generator: In each iteration, DocTer generates values for all required parameters and some optional parameters (for testing more diverse code). The probability for generating each optional parameter is `optional_ratio`.

The input generator generates one input for each iteration. Given the extracted constraints, DocTer generates a value for each parameter following the order determined by constraint dependencies (Section 2.4). For a conforming input, all generated arguments satisfy the extracted constraints for *structure*, *dtype*, *shape*, and *valid value*. If concrete values are specified (e.g., enumerated values) in the constraints, the input generator chooses from those values. Otherwise, it chooses a *dtype* from the list of *dtypes* specified in the constraints and creates a *shape* following the constraints. If the constraints do not specify valid *dtypes*, DocTer selects one from a default list of *dtype* described in Section 2.2. While the input generator is choosing *dtype* and *shape* for a parameter, it ensures they are generated according to the parameter dependencies, if any. For example, parameters often have matching dimension(s), so the input generator needs to ensure such shape consistency.

Once the *dtype* and *shape* are determined, the input generator generates an n-dimensional array with values satisfying the given *dtype*, *shape*, and the range as specified in the constraints, if any. Finally, the *structure* constraints are checked and satisfied. For example, if the generated value is 1-dimensional and the constraints explicitly specify the *structure* (e.g., tuple or list) for the parameter, the input generator converts the generated value accordingly.

To generate an invalid input, the input generator randomly selects one parameter and generates a value that violates one or multiple relevant constraints of that parameter. For all other parameters, DocTer generates their values in the same way as conforming inputs (i.e., conforming to all constraints).

Constraint-guided boundary-input generation: Boundary input values (e.g., 0 and `None`) tend to cause bugs due to off-by-one errors etc. [18, 24]. Thanks to the extracted constraints, DocTer generates boundary values that follow the constraints and boundary values that violate the constraints. For each API, DocTer picks one parameter with the probability of `mutation_p` to be mutated to one of the boundary cases. We consider six types of boundary mutators: one constraint-specific (boundary values of constraints) and five generics (`None`, zero, zero dimension, empty list, and empty string). As an example, the mutator "zero dimension" sets the size of one of the dimensions of the parameter's shape to 0 (e.g., it mutates a 3-D tensor of shape `[1,1,1]` to `[1,0,1]`).

Test case evaluator: The test case evaluator invokes the target function with the generated input. If a severe failure occurs, DocTer reports the input as a bug-triggering input. Specifically, DocTer returns those inputs causing a segmentation fault, floating-point

exception, abort, and bus error in the C++ backend. We exclude aborts caused by assertion failures in MXNet since MXNet uses those for exceptions. Crashes from the C++ backend (which handles computationally-intensive DL tasks) indicate severe problems.

3 EXPERIMENTAL SETUP

Data collection: We choose three popular DL libraries (TensorFlow 2.1.0, PyTorch 1.5.0, and MXNet 1.6.0) as test subjects. There are 144,541–854,900 words in the collected API documents. Among them, we consider 1008, 529, and 1021 relevant APIs for the three respective libraries. An API is irrelevant if it (1) is deprecated, (2) has no input argument, (3) cannot be parsed due to HTML syntactic errors and typos, (4) is a non-layer class constructor, or (5) has an API document without a "Parameter" description section. In total, 2,666 APIs are filtered out due to the five reasons above, 53.5% of which are due to reason (1). We list the break down in [13].

Rule construction and constraints extraction: DocTer applies three thresholds `MAX_SIZE`, `min_support`, and `min_confidence` to construct extraction rules (Section 2.3). We set `MAX_SIZE` to 7 to all three libraries. To select the best value for `min_support` and `min_confidence`, we conducted 5-fold cross-validation on the 30% annotated data and measure the quality of the extracted constraints. By selecting the best F1 score, we set `min_support` to 10, 10, 20, and `min_confidence` to 0.9, 0.7, 0.9 for TensorFlow, PyTorch, and MXNet, respectively.

Input generation and testing: We use Docker with Ubuntu 18.04, TensorFlow 2.1.0, PyTorch 1.5.0, and MXNet 1.6.0. For multi-dimensional arrays, DocTer generates shapes of 0–5 dimensions or as specified by the constraints. By trying different values of `optional_ratio` and `mutation_p` on 10% randomly sampled APIs, we choose `optional_ratio=0.2` and `mutation_p=0.4`.

Manual and execution time: The AC annotation (Section 2.2) takes 36 manual hours. DocTer takes 34 minutes to perform rule construction and constraints extraction for all libraries. On average, it takes DocTer 0.14 seconds to generate and test each input.

4 EVALUATION AND RESULTS

We answer four research questions (RQs): **RQ1:** How effective is DocTer in extracting constraints from DL API documentation? (Section 4.1) **RQ2:** How is DocTer compared to existing constraint-extraction approaches? (Section 4.2) **RQ3:** Can the extracted constraints enable DocTer to detect more bugs? (Section 4.3) and **RQ4:** How effective is DocTer in generating valid inputs? (Section 4.4)

4.1 RQ1: Effectiveness of constraint extraction

Approach: We apply DocTer to extract constraints in our subjects and study the number and quality of constraints. We randomly sample an extra of 5% (603) of input parameters (excluding the 30% AC annotated data for rule construction) to form an evaluation set. We manually annotate these parameters with concrete constraints to build the ground truth. The constraints extracted by DocTer are then compared against the evaluation set.

For each parameter, we consider all valid options for one category as one constraint. And the constraint for this category is correct iff all valid options are correctly extracted. For example, the parameter `size` of `tf.slice` can be either `int32` or `int64`. The extracted *dtype* constraint $p_T \in$ `{int32,int64}` is deemed correct, while $p_T \in$ `{int32}` is considered incorrect. If a parameter's document contains no constraints of the four categories, it is excluded from the precision and recall computation. While it is reasonable to include such no-constraint parameters in our calculation because DocTer can trivially extract nothing, the accuracy may be inflated if there is a large portion of such parameters. Among the sampled parameters, the numbers of no-constraint parameters are 29 (15.3%), 10 (10.8%), and 91 (28.4%) for TensorFlow, PyTorch, and MXNet, respectively (details in Extraction result section below).

Table 2: Quality of constraint extraction

	TensorFlow	PyTorch	MXNet	Total/Avg
# APIs with constr. extracted	911	498	1,006	2,415
# constr. extracted	5,908	3,201	6,926	**16,035**
# constr. per API: *Avg (Min-Max)*	6.5 (1-51)	6.4 (1-33)	6.9 (1-111)	6.6 (1-65)
# evaluated param.	190	93	320	603
# evaluated param. with constr.	161	83	229	473
# evaluated constr.	350	170	363	883
Precision/Recall/F1 for All (%)	90.0/74.8/81.7	78.4/77.4/77.9	87.9/82.4/85.1	**85.4**/78.2/81.6
Precision/Recall/F1 for *dtype* (%)	93.0/82.3/87.3	78.1/79.4/78.7	92.9/81.9/87.0	88.0/81.2/84.3
Precision/Recall/F1 for *structure* (%)	78.9/88.2/83.3	85.7/90.0/87.8	91.7/90.2/92.4	85.4/89.5/87.8
Precision/Recall/F1 for *shape* (%)	89.1/74.5/81.2	80.0/76.9/78.4	76.1/79.8/77.9	81.7/77.1/79.2
Precision/Recall/F1 for *valid value* (%)	87.5/47.7/61.8	66.7/60.0/63.2	90.0/60.0/72.0	81.4/55.9/65.7

We use the standard metrics precision, recall, and F1 score of the extracted constraints of the sampled parameters for each constraint category. *Precision* is the percentage of the correctly extracted constraints (i.e., extracted constraints that match the ground-truth) over the number of all extracted constraints. *Recall* is the percentage of correctly extracted constraints over the total number of all ground truth constraints. *F1* is the harmonic mean of precision and recall.

Extraction results: Table 2 shows the quality of extracted constraints. In total, DocTer extracts 16,035 constraints automatically from the three libraries (row *#constr. extracted*). Specifically, TreeMiner [55] collects 873, 426, and 321 frequent parse subtrees from the three libraries respectively with the corresponding `min_support`. Then DocTer constructs rules with 665, 398, and 275 subtrees. The remaining subtrees do not constitute any rules because no AC is associated with the subtree with large enough conditional probability. Using these rules, DocTer extracts on average 6.6 constraints per API for all three libraries (row *#constr. per API: Avg (Min-Max) Table 2)*. Overall, DocTer can extract constraints from 90.4%, 94.1%, and 98.5% of relevant APIs (details in Section 3) for TensorFlow, PyTorch, and MXNet, respectively.

For each library, Table 2 shows the number of parameters in the evaluation set (row *#evaluated param.*), the number of the parameters in the evaluation set with at least one constraint (row *#evaluated param. with constr.*), the number of constraints manually labeled in the evaluation set (row *#evaluated constr.*). The *Total/Avg* column shows the total number of parameters and constraints in the evaluation set, and the average precision, recall, and F1 score.

Overall, DocTer achieves a high precision (85.4%) and recall (78.2%) of constraint extraction across all three subjects. DocTer is quite effective in extracting constraints for *dtype* and *structure* with F1 score over 80%. It is less effective when extracting constraints for *valid value*. The reason is that sentences that describe constraints for *valid value* are not as common in the annotated data compared with other categories, e.g., *structure*, and thus DocTer misses some patterns given the `min_support` and `min_confidence` thresholds. For

Table 3: Number of rules and constraints

Category	TensorFlow		PyTorch		MXNet		Total	
	Rules	Constraints	Rules	Constraints	Rules	Constraints	Rules	Constraints
dtype	405	2,392	114	1,163	196	2,272	715	5,827
structure	230	1,305	151	890	78	2,466	459	4,661
shape	306	1,825	282	852	173	1,699	761	4,376
valid value	97	386	22	296	11	489	130	1,171
All	665	5,908	398	3,201	275	6,926	1,338	16,035

example, when analyzing the sentence "*Only 'zeros' is supported for quantized convolution at the moment*", DocTer misses the *valid value* constraint $p_0 \in \{\texttt{'zeros'}\}$ because the pattern "*Only ... is/are supported*" is not frequent enough in the annotated dataset and did not pass the set thresholds. In Fig. 1, DocTer misses the constraint that parameter `ksize` can also be a single integer due to the same reason. With the extracted incomplete constraints (i.e., `ksize` is a list of integers), DocTer still detects the bug. This confirms that to detect real-world bugs effectively, one does not need to have complete constraints [52]. We choose the thresholds as a trade-off between precision and recall, and one can choose lower thresholds for a better recall.

To show the impact of our rule extraction design (Section 2.3), we conduct an ablation study, where we do not use the conditional probabilities (Eq. 2) when constructing rules (the set R) and instead set the `min_confidence` to 0 (instead of the settings in Section 3). Under these settings, any frequent subtree in $trees(D_f)$ and any AC in A_f that has more than one co-occurrence will be considered as a rule. As a result, this version of DocTer without the conditional probabilities extracts 1,621 imprecise rules and extracts constraints with an F-1 of 27.9% on the same evaluation set (Table 2). In contrast, our DocTer, with the conditional probabilities, constructs 1,338 rules (Table 3) and extracts constraints with a much higher F-1 of 81.6% (Table 2) on the same data.

Overall, DocTer extracts tens of thousands of correct constraints for these libraries, which enables the generation of valid inputs for detecting 94 bugs. We show the breakdown of the number of rules and constraints extracted for all three libraries in Table 3. Note that a subtree can be mapped to multiple categories of ACs.

Sensitivity study: Since one can choose to annotate fewer parameters to save manual effort at the cost of a reduced F-1 score, we quantify the trade-off between the effectiveness of our approach (measured by F-1) and manual effort, which is measured by the amount of parameters to annotate. Specifically, we evaluate the F-1 scores of our approach by using different amounts of the annotated data, i.e., 5%, 10%, and 30% of parameters. The 5% of parameters are a subset of the 10% of the parameters, which is a subset of the 30% of the parameters. DocTer achieves an overall F-1 score of 66.0% with just 514 annotated parameters (5% of the parameters), 73.9% with 1,028 annotated parameters (10% of parameters), and 81.6% with 3,086 annotated parameters (30% of the parameters).

Generality of rules: To evaluate the generality *across libraries*, we apply the rules constructed from TensorFlow and MxNet to the documentation of PyTorch, and get the constraints with precision, recall, and F1 of 87.9%, 70.3%, and 78.1%. In addition, with the rules DocTer constructed from all three libraries, we extract 2,312 constraints from 223 scikit-learn APIs' documents. We manually inspect the extracted constraints on 5% (59) randomly sampled parameters of scikit-learn APIs. DocTer achieves a precision/recall/F1 of 71.3/66.1/68.6%. The results suggest that *DocTer can be applied to new libraries completely automatically without requiring annotating any documents of the new libraries.*

To evaluate the generality *across versions* of the same library, we apply the rules that DocTer constructed from TensorFlow v2.1.0, PyTorch v1.5.0, and MXNet v1.6.0 to six versions: two more recent versions from each library respectively (TensorFlow v2.2.0 and v2.3.0, PyTorch v1.6.0 and v1.7.0, and MXNet v1.7.0 and 1.8.0). DocTer extracts 59,936 constraints from 7,684 APIs containing 580,187 words. For evaluation, we randomly sample 603 parameters (same number of parameters as the evaluation in Table 2) that are either with updated descriptions or newly added to the more recent versions. The results show that the rules that DocTer constructed are general across versions and can extract constraints from other versions with a precision/recall/F1 of 81.9/77.7/79.7%.

4.2 RQ2: Comparison with existing approaches

We compare DocTer with grep and state-of-the-art constraint extraction approaches (e.g., Jdoctor [16]).

Comparison with grep: While it may appear to be straightforward to use a *grep-like* technique (i.e., matching existing keywords in documents) to extract constraints, such a technique can only identify relevant API document sentences. DocTer, on the other hand, extracts concrete constraints automatically. The *grep-like* approach could assign a constraint to a match, e.g., if a sentence contains the keyword "integer", the corresponding parameter would be assigned the constraint $p_T \in \{\texttt{int}\}$. We implement this approach by searching in the documents for keywords of *dtype* constraints (e.g., "int" and "integer") , and *structure* constraints (e.g., "list" and "tensor"). We manually collect such keywords in the API documents. This approach misses 47.8% of the constraints that DocTer extracts, i.e, all *shape*, all *valid value*, 33% of *dtype*, and 4% of *structure* constraints.

Comparison with existing constraint-extraction techniques: We compare DocTer with the state-of-the-art constraint-extraction techniques, including Jdocter [16], DASE [52], Zhou et al. [60], and Advance [31]. We exclude C2S [56] because it requires formal specifications (JML [1]), which is unavailable for the three libraries. We exclude Pytype [7] because it cannot analyze across Python and C++ code, therefore, cannot precisely infer types for DL libraries. Jdocter [16], DASE [52], and Advance [31] can only extract constraints for *valid value* (e.g., range). With the assumption that they can extract all *valid value* constraints correctly, the best (upper bound) recall that these tools can achieve is 11.9%. Aside from *valid value*, Zhou et al. [60] is able to extract specifications for type restrictions. However, we found that their heuristics that can be applied to DL document, e.g., "[something] be not [SpecClassName]", extract at most 28 constraints (0.2% of DocTer constraints). This results in their best recall of 12.3%, while DocTer has a recall of 78.2%. Overall, *shape* constraints are DL-specific that DocTer extracts while existing techniques do not consider. The results show that DocTer complements existing constraint-extraction techniques by extracting DL-specific constraints automatically.

Table 4: Number of verified new / new / all bugs (buggy APIs)

Approach		TensorFlow	PyTorch	MXNet	Total
Baseline		22 / 26 / 41 (79)	6 / 6 / 7 (8)	6 / 9 / 11 (21)	34 / 41 / 59 (108)
DocTer	All	31 / 38 / 61 (114)	13 / 13 / 18 (28)	10 / 12 / 15 (32)	54 / 63 / **94 (174)**
	CI	21 / 28 / 47 (93)	11 / 11 / 14 (23)	10 / 12 / 14 (27)	42 / 51 / 75 (143)
	VI	28 / 32 / 51 (83)	13 / 13 / 18 (25)	8 / 10 / 12 (27)	49 / 55 / 81 (135)

4.3 RQ3: Bug detection results

Approach: We demonstrate the effectiveness of DocTer's constraint extraction using the constraints to guide input generation to detect bugs in API documents and library code. For documentation bugs, DocTer detects inconsistencies within API documents when extracting constraints, which will be discussed later in this section. For library code bugs, we use all 16,035 (Table 2) constraints extracted by DocTer to generate inputs for API functions that have at least one extracted constraint. Table 2 shows the numbers of these API functions (row *#APIs with extracted constr.*). We set `maxIter` to 2,000. For each API function, DocTer generates 2,000 test inputs (1,000 conforming and 1,000 violating inputs), evaluates them, and returns bug-triggering inputs that cause serious failures (details in Section 2.5). We manually examine those bug-triggering inputs to check if they reveal real bugs. For those inputs that still trigger the same failures in the nightly version, we report the bugs to the developers.

We implement an unguided input generation tool as the *baseline*. The only difference between DocTer and the baseline is that the baseline has no knowledge of constraints. Specifically, the baseline generates 2,000 random inputs for parameters without any constraint knowledge. For a fair comparison, we convert the generated array inputs to tensors assuming that the baseline minimally knows which input arguments should be tensors. Without this conversion, non-tensor input arguments are trivially rejected by PyTorch and MXNet, thus very ineffective in exercising the code in depth.

The extracted constraints can be used together with other input generation tools to improve their testing effectiveness. In this paper, we choose to implement our own baseline instead of using existing fuzzers [3, 5, 43] such as AFL [3] for practical reasons. These fuzzers cannot test Python code: the most popular language for DL. Moreover, these fuzzers require code coverage, which is currently unavailable across Python and C++. Instead of code coverage, DocTer uses constraints extracted from documents to guide the testing of both Python and C++ code, by generating inputs for the Python API functions, in which C++ code is invoked. In addition, existing fuzzers [3, 5, 43] generate inputs in the format of a sequence of byte arrays. Randomly mutating some bytes is unlikely to generate valid DL-specific inputs. Our baseline is similar to AFL with two enhancements: (1) knowledge of tensors and (2) automatically testing Python and C++ code.

Bugs in libraries code: Table 4 presents the number of *verified new bugs, new bugs, all bugs*, and *buggy APIs (in "()")* found by the baseline and DocTer. A bug is verified if it has been fixed or confirmed by the developers. A new bug refers to a previously unknown bug that we reported. The unverified new bugs are reproducible and waiting for developers' responses. We count one bug for each required fixing location.

DocTer detects 94 bugs, including 63 previously unknown bugs, 54 of which have been verified by the developers (49 fixed and 5 confirmed). Of the 49 fixed bugs, 19 are fixed in C++, 11 are fixed in Python, 7 are fixed in both, and 12 is fixed silently after we reported. The 94 bugs cause 174 APIs to fail because one bug can cause failures in multiple APIs but are fixed in one location. We count them as 94 instead of 174 bugs. Of the 174 buggy APIs, 12 have parameters with constraint dependencies. DocTer has also detected 31 (94 − 63) known bugs that have been fixed in the nightly versions.

The baseline detects only 59 bugs with 41 new bugs causing 108 APIs to crash. DocTer detects 52 bugs that the baseline cannot while missing 7 bugs found by the baseline due to the randomness of the input generation process.

False positives: Only 3 out of 66 newly reported bugs receive "won't fix" responses from the developers. They claim such inputs are not supported, which is not stated in the document. We do not count these 3 bugs in our results.

DocTer uses the automatically extracted constraints without any manual examination. It is possible that documents themselves are incorrect or incomplete, causing incorrect constraints to be extracted, leading to false positives, where the code is correct, but the API documentation is incorrect. Since we focus on severe bugs such as crashes, all detected bugs are in library code bugs, as well said by a developer after we reported a crash bug "*A segmentation fault is never OK and we should fix it with high priority.*"

Conforming and violating inputs: DocTer generates both conforming inputs (CIs) and violating inputs (VIs). Rows "CI" and "VI" in Table 4 present the breakdown of the bug detection for CIs and VIs with `conform_ratio = 50%`. We find that DocTer is insensitive to conform_ratio. When it is between 20%–60% (with a 10% increment): the number of detected bugs differs by at most one. Thus, we use conform_ratio = 50% as the default ratio to be more general.

The results show that the CIs alone, with only 50% (1,000) of the number of test inputs of the baseline (2,000), finds more bugs (75 bugs) than the baseline (59 bugs), and the VIs alone (with 50% of the test inputs) finds more bugs (81 bugs) than the baseline. We manually verify the generated CIs and VIs: out of 75 CI bugs we found, 57 of them are caused by valid inputs conforming to the ground truth constraints. The rest of the CI bugs are caused by invalid inputs generated by conforming to inaccurate constraints; out of 81 VI bugs we found, all of them are caused by invalid inputs violating the ground truth constraints.

Many bugs are detected by both CIs and VIs (comparing the "All" row with the "CI" and "VI" rows in Table 4) because DocTer violates the constraints of one parameter only when generating VIs. When a crash is caused by one of the conforming parameters of a VI, it is likely to be triggered by a CI also. However, both CIs and VIs trigger bugs in unique APIs, thus both are effective in finding bugs.

Without the constraints, a baseline is much worse than the results from any of the ratio setups. Table 4 shows that a *key contribution of our work is the ability to extract constraints from documents.* One cannot choose to focus on valid or invalid inputs without knowing the definition of valid inputs for an API. DocTer enables this choice since it extracts input constraints automatically.

Impact of fuzzer's nondeterminism: Since the fuzzing process is non-deterministic, we investigate the impact of this nondeterminism to ensure the validity of our results, i.e., the reported improvement of DocTer is not due the randomness in the fuzzer. Our results suggest that it is statistically significant that DocTer (which is guided by constraints) outperforms a baseline fuzzer. Specifically, we repeat the fuzzing experiment (with the same set of extracted constraints) eight times. For each run, we generate 2,000 test inputs for the baseline and 2,000 test inputs for DocTer for all APIs from the three libraries. In each run, we use the same random seed for both the baseline and DocTer. Different runs use different seeds. Since it requires significant manual effort to inspect the detected bugs from all eight runs, which are 2,301 API crashes to examine, we use the number of buggy APIs (i.e., the number of APIs that crash) to indicate the fuzzers' effectiveness. Overall, among the eight runs, DocTer on average detects 172.0 buggy APIs while the baseline on average detects 115.6 buggy APIs. We perform the Mann-Whitney U-test and confirm the improvement is statistically significant with a p-value of 0.0004 and the Cohen's d effect size of 8.96 (effect size more than 2.0 is huge). The detailed results are in [13].

Bugs in API documents: DocTer detects three types of documentation bugs: (1) formatting bugs (e.g., indentation issue); (2) signature-description mismatch (the description refers to parameters that are not specified in the API signature); and (3) unclear constraint dependency (Section 2.4). DocTer detects 43 previously unknown documentation bugs in 46 APIs (11 formatting bugs, 29 signature-description mismatches, 3 unclear constraint dependencies). Majority (39 of 43) are fixed or confirmed after we report, indicating that DocTer detects documentation bugs that developers care to fix.

Bug examples: We present three bugs detected by DocTer that the baseline fails to detect. All of them have been fixed by developers after we report them. **Bug 3** is also reported as a security vulnerability CVE-2020-15265.

Bug 1: The previously unknown bug in TensorFlow `tf.nn.max_pool3d` discussed in the Introduction (Fig. 1).

Bug 2: In the API `tf.image.combined_non_max_suppression`, DocTer detects a previously-unknown bug and triggers a memory overflow by passing a large value of 311452676770467072 for the parameter `max_total_size`. To successfully trigger this bug, DocTer needs to generate correct shaped values for parameters `boxes` and `scores`. Specifically, the parameter `boxes` needs to be 4-D with the size of the last dimension equals to 4 while the parameter `scores` needs to be 3-D. DocTer also needs to follow the dependencies between those two parameters – the sizes of the first two dimensions of `boxes` and `scores` need to be the same. DocTer does this by extracting relevant *shape* constraints and the dependencies correctly from the API document. Without such knowledge, random input generation fails to produce valid input for `boxes` and `scores` to trigger this bug.

Bug 3: DocTer triggers a segmentation fault bug in the TensorFlow API `tf.quantization.quantize_and_dequantize` using an `input` tensor of any shape with an out-of-bound `axis` value (i.e., the value of `axis` is larger than the number of `input` dimensions). After we report the bug, TensorFlow developers report this as a security vulnerability CVE-2020-15265 to the national vulnerability database (NVD). The extracted constraints enable DocTer to trigger this bug by ensuring the generation of many valid input values for all parameters of this API other than the `axis` values. For example, the extracted constraints contain the boolean type for parameters `narrow_range`, `range_given`, and `signed_input` and the valid values ("`HALF_TO_EVEN`" and "`HALF_UP`") for parameter `round_mode`, which help DocTer generate valid values for these parameters. In contrast, since the baseline is unaware of these constraints, the baseline generates invalid values for these parameters, which are rejected by TensorFlow's input validation code, therefore avoiding exposing this security vulnerability.

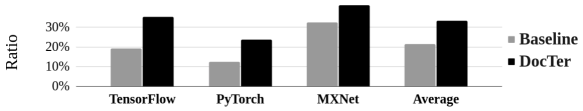


Figure 3: Ratio of passing inputs

4.4 RQ4: Valid-input generation results

Approach: As discussed in the Introduction, generating valid inputs is essential to exercise the core functionality of the API function. While DocTer attempts to generate CIs, these CIs may still be invalid if the constraints extracted are incorrect or incomplete. We study the percentage of generated CIs that are valid inputs. We compute the ratio out of 1,000 CIs (`confirming_ratio = 50%`) with the first 1,000 baseline inputs for each API function. Since manually examining the validity of all inputs is impractical and the validity checking of mature projects (e.g., our subjects) is generally reliable, we make a reasonable approximation by counting the number of passing inputs whose executions terminate normally.

Results: Fig. 3 presents the ratio of passing inputs for each subject and the average. On average, DocTer achieves 33.4% ratio of passing inputs, which outperforms the baseline (21.5%) by generating 55.3% more passing inputs. The results suggest that DocTer is more effective in generating valid inputs than the baseline to detect more bugs. Although DocTer outperforms the baseline, the ratio of passing inputs is still low (33.4%), because API documents are often incomplete. DocTer might convince developers to write more complete documents since documents can help them find bugs.

5 THREATS TO VALIDITY

Complex constraints: DocTer does not work with complex constraints that require a nested structure or indirect dependency with the constraints of another parameter. However, these complex constraints are uncommon in DL libraries (appeared in only 6.4% of our sampled parameters).

Testing Python and C++ code: Since DL libraries' core computations are in C++, it may appear to be more reasonable to directly test C++ code. However, since Python APIs are the most popular for DL, testing them is testing the common use cases. DocTer tests Python APIs which invoke the computations in C++, so DocTer finds bugs in both Python (26 bugs) code and C++ (18 bugs) code.

Manual annotations: There is a one-time cost of up to 36 man hours of manually annotating 30% of parameters with their ACs (Section 2.2). Since the DocTer-generated rules are applicable to

other libraries and versions (Section 4.1), this one-time cost is reasonable. Such manual annotation cost is widely accepted in other domains (e.g., supervised learning). Moreover, to minimize biases with the manual annotation, our process involves three co-authors. Two authors independently annotate with 98.2% agreement. All disagreements are resolved with a third author to reach a consensus.

6 RELATED WORK

DocTer is the first technique to extract DL-specific constraints from API documentation, and the first DL library testing technique that is guided by such input-constraints.

Constraint extraction: Existing constraint-extraction techniques are insufficient for extracting DL-specific constraints [7, 16, 31, 46, 52, 56, 60], because they miss most of the DL-specific constraints, cannot analyze across Python and C++ code, or requires formal specifications. Many existing techniques [16, 22, 46, 52, 53] use a handful of manually-designed rules to extract constraints. Instead, DocTer uses subtree matching to automatically construct rules to extract constraints.

Analyzing software text to detect bugs: Prior work leverages documents [31] and comments [44–46, 60] to detect inconsistency bugs between code and its specifications. Some prior work translates software specifications into assertions [30] and oracles [22, 34]. Different from these techniques, DocTer uses frequent subtree mining and association rule learning to extract constraints from API documents to guide input generation for testing DL libraries.

Testing DL libraries and fuzzing: The constraints extracted may be used to improve existing testing techniques. The DL library testing techniques focus on addressing the test oracle challenge, by using differential testing [18, 24, 41, 42, 49, 51] or oracle approximation [35, 59]. DocTer uses crashes instead and addresses the challenge of obtaining input constraints automatically.

Existing techniques are designed to detect specific types of bugs such as shape-related (e.g., tensor shape mismatch) [18, 28], numerical [18, 24] (e.g., returns `NaN/Inf`), decreased accuracy [18], and performance [47]. On the other hand, DocTer finds general bugs that lead to serious crashes.

TensorFlow developers use OSS-Fuzz [6] along with libFuzzer [5] to test only 19 TensorFlow's C++ API functions. It requires developers to manually encode test inputs from the byte-arrays returned by libFuzzer. This would take a prohibitive amount of manual effort to test the 2,415 APIs that DocTer tests.

Fuzzers [3, 5, 9] have been adopted to test non-DL libraries [29, 40]. They would not work well for DL libraries (Section 4.3). Since Randoop [37] works only for a statically-typed language (e.g., Java), it would fail to create valid dynamically-typed objects for Python (the most popular language for DL [8]).

Testing DL models: Many fuzzing techniques test the robustness of DL *models* instead of DL *libraries* [20, 23, 25, 36, 48, 50, 54]. DocTer tests DL libraries since testing DL models alone is insufficient as DL libraries contain bugs [26, 27, 41].

7 CONCLUSION

We propose DocTer, which features a novel method to derive general rules to translate API documents to precise parameter constraints. We apply these rules to popular DL libraries to extract a large number of DL-specific constraints. We use the constraints to guide the input generation of DL API functions. The constraints enable DocTer to generate valid and invalid inputs to detect more bugs in code and documents.

ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their invaluable feedback. The research is partially supported by NSF 1901242 award.

REFERENCES

[1] 1999. *The Java Modeling Language (JML).* "https://www.cs.ucf.edu/~leavens/JML/examples.shtml"
[2] 2004. *Beautiful Soup.* https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[3] 2013. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
[4] 2014. Universal Dependencies. https://universaldependencies.org/
[5] 2015. libFuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html
[6] 2016. *OSS-Fuzz.* https://github.com/google/oss-fuzz
[7] 2016. *pytype.* "https://github.com/google/pytype"
[8] 2017. What is the best programming language for Machine Learning? https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7.
[9] 2019. *FuzzFactory: Domain-Specific Fuzzing with Waypoints.* "https://github.com/rohanpadhye/fuzzfactory"
[10] 2019. incubator-mxnet. https://github.com/apache/incubator-mxnet/blob/1.6.0/python/mxnet/ndarray/ndarray.py#L64-L74
[11] 2019. torch.Tensor. https://pytorch.org/docs/1.5.0/tensors.html
[12] 2020. tf.dtypes.DType. https://www.tensorflow.org/versions/r2.1/api_docs/python/tf/dtypes/DType
[13] 2022. DocTer's Supplementary Material. https://github.com/lin-tan/DocTer
[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16).* 265–283.
[15] Steven Bird, Edward Loper, and Ewan Klein. 2009. *Natural Language Processing with Python.* O'Reilly Media Inc.
[16] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 242–253.
[17] 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. arXiv:1512.01274 [cs.DC]
[18] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. *(ESEC/FSE 2018).* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3236024.3236057
[19] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (feb. 2013), 276 –291.
[20] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20).*
[21] P. Godefroid, A. Kiezun, and M. Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation.* 206–215.
[22] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis.* 213–224.
[23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018).* ACM, New York, NY, USA, 739–743. https://doi.org/10.1145/3236024.3264835
[24] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).*
[25] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1158–1161.

[26] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of 42nd International Conference on Software Engineering (ICSE '20)*. ACM.

[27] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[28] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs.. In *ECOOP 2020*.

[29] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage.. In *ASE*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485.

[30] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. 2014. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 785–790.

[31] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. 2020. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1837–1852.

[32] R. Majumda and R. Xu. 2007. Directed Test Generation Using Symbolic Grammars. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 134–143.

[33] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[34] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 188–199.

[35] M. Nejadgholi and J. Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 785–796.

[36] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 4901–4911.

[37] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[40] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation.. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 697–710.

[41] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[42] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software. In *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*.

[43] Robert Swiecki. 2015. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URl: https://github. com/google/honggfuzz* (2015).

[44] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*Icomment: Bugs or Bad Comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, New York, NY, USA, 145–158. https://doi.org/10.1145/1294261.1294276

[45] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1985793.1985796

[46] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 260–269. https://doi.org/10.1109/ICST.2012.106

[47] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 189–199.

[48] Sakshi Udeshi and Sudipta Chattopadhyay. 2019. Grammar Based Directed Testing of Machine Learning Systems. *CoRR* abs/1902.10027 (2019). arXiv:1902.10027

[49] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering discrepancies in numerical libraries.. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis (ISSTA 2020)*. 488–501. https://doi.org/10.1145/3395363.3397380

[50] Haohan Wang, Da Sun, and Eric P Xing. 2019. What if we simply swap the two text fragments? a straightforward yet effective way to test the robustness of methods to confounding signals in nature language inference tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7136–7143.

[51] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[52] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 620–631.

[53] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2013. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*. 1421–1432.

[54] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 146–157. https://doi.org/10.1145/3293882.3330579

[55] Mohammed Javeed Zaki. 2005. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE transactions on knowledge and data engineering* 17, 8 (2005), 1021–1035.

[56] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[57] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM.

[58] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis (ISSTA 2018)*. 129–140. https://doi.org/10.1145/3213846.3213866

[59] W. Zheng, W. Wang, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie. 2019. Testing Untestable Neural Machine Translation: An Industrial Case. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 314–315.

[60] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. 2018. Automatic detection and repair recommendation of directive defects in Java API documentation. *IEEE Transactions on Software Engineering* 46, 9 (2018), 1004–1023.