

Documentation-Guided Fuzzing for Testing Deep Learning API Functions

by

Yitong Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

© Yitong Li 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

This thesis is based on the following joint work:

Yitong Li, Danning Xie, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions.

The contributions of the author of this thesis include:

- Designing and implementing the Fuzzing component of DocTer, and conducting necessary experiments to detect the library bugs
- Investigating and confirming the detected library bugs
- Extracting the ground truth of the sampled constraints for evaluation
- Performing variance analysis to determine the ratio of conforming and violating inputs for best detection performance
- Writing of this thesis

Since this work would not have been possible without the contributions of all the coauthors, the author uses an editorial “we”, rather than “I”, even when describing work performed by the author of this thesis individually.

Abstract

Widely-used deep learning (DL) libraries demand reliability. Thus, it is integral to test DL libraries’ API functions. Despite the effectiveness of fuzz testing, there are few techniques that are specialized in fuzzing API functions of DL libraries.

To fill this gap, we design and implement a fuzzing technique called DocTer for API functions of DL libraries. Fuzzing DL API functions is challenging because many API functions expect structured inputs that follow DL-specific constraints. If a fuzzer is (1) unaware of these constraints or (2) incapable of using these constraints to fuzz, it is practically impossible to generate valid inputs, i.e., inputs that follow these DL-specific constraints, to explore deep to test the core functionality of API functions. DocTer extracts DL-specific constraints from API documents and uses these constraints to guide the fuzzing to generate valid inputs automatically. DocTer also generates inputs that violate these constraints to test the input validity checking code. To reduce manual effort, DocTer applies a sequential pattern mining technique on API documents to help DocTer users create rules to extract constraints from API documents automatically.

Our evaluation on three popular DL libraries (TensorFlow, PyTorch, and MXNet) shows that DocTer’s accuracy in extracting input constraints is 82.2–90.5%. DocTer detects 46 bugs, while a baseline fuzzer without input constraints detects only 19 bugs. Most (33) of the 46 bugs are previously unknown, 26 of which have been fixed or confirmed by developers after we report them. In addition, DocTer detects 37 inconsistencies within documents, including 25 fixed or confirmed after we report them.

Acknowledgements

I would like to thank my advisor Professor Lin Tan and Professor Michael Godfrey for providing immense guidance and support. They have taught me the methodologies to carry out the research and shared invaluable visions to steer the direction of this work.

I would like to thank Hung Viet Pham and Thibaud Lutellier for providing insightful suggestions as colleagues, and generous support as friends.

I would especially like to thank my family and my girlfriend for their love and encouragement over the years.

Table of Contents

List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Our approach	3
1.2 Contributions	5
2 Approach	6
2.1 Challenges and overview	6
2.2 Pattern miner	8
2.2.1 API document collection and preprocessing	8
2.2.2 Sequential pattern mining	9
2.3 Rule construction	9
2.4 Constraint extractor	11
2.4.1 Constraint dependencies	11
2.4.2 Documentation bug detection	12
2.5 Fuzzing process	12
2.5.1 Input generator	12
2.5.2 Test case evaluator	13

3	Experiments and Results	14
3.1	Data collection	14
3.2	Constraint extraction	14
3.3	Fuzzing	15
3.4	Constraint extraction results	15
3.4.1	Approach	15
3.4.2	Extraction results	16
3.4.3	Generality of rules	17
3.5	Bug detection results	18
3.5.1	Approach	18
3.5.2	Bugs in library code	19
3.5.3	Conforming and violating inputs	20
3.5.4	Bugs in API documents	21
3.6	Bug examples	22
3.6.1	Bug 2	22
3.6.2	Bug 3	22
3.6.3	Discussion	22
3.7	Valid-input generation results	23
3.7.1	Approach	23
3.7.2	Results	23
4	Threats to Validity	25
4.1	Practicality of generated inputs	25
4.2	Complex constraints	25
4.3	Manual rule construction	25
4.4	Python test inputs	26

5	Related Work	27
5.1	Testing DL libraries	27
5.2	Unit test generation and fuzzing	28
5.3	Analyzing software text to detect bugs	28
6	Future Work	29
6.1	Complex constraints	29
6.2	Input minimization	29
6.3	Feedback-directed fuzzing	29
6.4	Use address sanitizer	30
7	Conclusion	31
	References	32

List of Figures

1.1	Documents of PyTorch API function <code>grid_sample</code>	2
2.1	Overview of DocTer	6
3.1	Ratio of passing inputs	23

List of Tables

2.1	Rule examples and the extracted constraints	10
3.1	Quality of constraint extraction	16
3.2	Rule overlap across the three libraries	18
3.3	Number of verified new / new / all bugs found by DocTer and the baseline	19

Chapter 1

Introduction

With growing interests in building intelligent systems using Machine Learning (ML) including Deep Learning (DL) techniques, various libraries (e.g., TensorFlow [11] and PyTorch [87]) have been released to allow developers to easily integrate ML algorithms in their applications. These libraries have made developing ML models efficient and convenient. However, ML libraries contain bugs [126, 52, 127, 51, 90], which hurt not only the development but also the accuracy and speed of the ML models. Therefore, a DL library needs to be well-tested for better reliability. A standard practice of testing these API functions is to pass various input values to these functions and inspect any unexpected behaviors triggered by the given inputs. Fuzzing is a scalable and practical testing technique for this purpose. Fuzzing provides random data as test inputs to a program and monitors if the inputs trigger any error in the program such as a crash [94]. Fuzzing a DL library's API functions is challenging because many of these API functions expect structured inputs that follow DL-specific constraints. If a fuzzer is (1) unaware of these constraints or (2) incapable of using these constraints to fuzz, it is practically impossible to generate *valid* inputs (i.e., inputs that follow these DL-specific constraints) to explore deep to test the core functionality of DL API functions.

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 a specific *data structure* such as lists, tuples, and tensors to perform numerical computations. For example, the PyTorch API function `torch.nn.functional.grid_sample` has two parameters, `grid` and `padding_mode` (other parameters are omitted for demonstration purpose). The former has to be a tensor, while the latter has to be a string, as dictated by its API document shown in Fig. 1.1a. A *tensor* is represented using an n-dimensional array, where *n* is a non-negative integer.

torch.nn.functional.grid_sample

```
torch.nn.functional.grid_sample(..., grid: torch.Tensor, ...,  
padding_mode: str = 'zeros', ...) → torch.Tensor
```

[SOURCE]

Parameters

- **grid** (*Tensor*) – flow-field of shape (...) (4-D case) or (...) (5-D case)
- **padding_mode** (*str*) – padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'

(a) API Document

```
grid: ndim:{4,5}  
  
padding_mode: dtype:{string},  
default: zeros,  
enum:{border,reflection,zeros}
```

(b) Extracted constraints

```
grid:  
  torch.tensor([[[[ 2.3e+38, 0]]]])  
padding_mode:  
  'reflection'
```

(c) Bug-triggering input

```
- return minimum(Vec(max_val), maximum(in, Vec(0)));  
+ // ... in order to clamp Nans to zero  
+ return clamp_max(Vec(max_val), clamp_min(Vec(0), in));
```

(d) Bug fix in GridSamplerKernel.cpp

Figure 1.1: Documents of PyTorch API function `grid_sample`

Any input that cannot be interpreted as a tensor (e.g., a `String`) 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 the API function. To test `grid_sample`’s core functionality, a fuzzer needs to generate a tensor object for the `grid` 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 is often insufficient to pass 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. 1.1a, the *dtype* of the parameter `padding_mode` 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. In Fig. 1.1a, the parameter `grid` should be a tensor of either 4 dimensions or 5 dimensions. Any inputs that do not follow these *dtype* or *shape* requirements are rejected

by the API function. Such inputs would exercise only the input validity checking code of the API function and fail to test the core functionality of the API function.

There is a lack of techniques that are specialized in fuzzing API functions of DL libraries. First, fuzzing techniques in the DL domain have been mostly used to test the robustness of DL *models* instead of DL *libraries* by finding adversarial inputs (e.g., images or natural language texts) for the models [77, 43, 120, 117, 133, 111, 110, 76, 113, 106, 131, 67, 48, 34]. As discussed earlier, testing DL models alone is insufficient, as a DL library itself contains bugs [126, 52, 127, 51, 90], which hurt the accuracy and speed of the entire DL system [90].

Second, general-purpose fuzzers that support API fuzzing cannot effectively generate valid inputs for DL API functions. Fuzzers such as AFL [1], Honggfuzz [100], and libFuzzer [5] generate inputs in the format of a sequence of byte arrays. Thus, randomly mutating some bytes in the input is unlikely to generate valid DL-specific data structures (i.e., tensors) following their properties. Other test generation tools such as Randoop [82] generate a sequence of function calls to create various states of objects under which the function is executed. However, this approach works only for a statically-typed language (e.g., Java) where each variable has a static type. It would fail to create valid objects in Python, which is the most popular language used for DL libraries [9], because Python is a dynamically-typed language where variable types are unknown until runtime.

1.1 Our approach

To fill this gap and tackle these challenges, we develop a fuzzing technique called *DocTer*, which extracts constraints from API documentation to guide the generation of test inputs for DL API functions. Since DL API functions are dominantly written in more than one programming language, e.g., Python for the user interface code and C++ for the core matrix calculations, DocTer, which generates inputs for the Python API functions, tests both the Python code and the C++ code. 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 tedious and inefficient. In addition, since these documents are constantly evolving, it is undesirable and error-prone to manually analyze them each time when the documents update which can be as frequent as every commit. To address these challenges, we leverage *sequential pattern mining* [42, 46] to mine frequently occurring patterns in API documents and manually transform them into rules to extract constraints automatically.

DocTer first analyzes free-form API documentation (e.g, Fig. 1.1a) to extract input constraints (e.g., Fig. 1.1b). DocTer uses these constraints to guide the fuzzing so that it

generates a valid input (e.g, Fig. 1.1c) that satisfies the constraints. DocTer then evaluates the generated test input by checking if it runs successfully without failures, e.g., crashes. If a failure occurs with a valid input, it is highly likely that the generated test has manifested a bug in the implementation of the API’s core functionality.

Fig. 1.1d shows a previously unknown bug detected by DocTer in PyTorch, and its patch that the PyTorch developers committed, after we reported the bug to them. According to the document in Fig. 1.1a, the shape of parameter `grid` is either 4-D or 5-D `tensor`. Following the extracted constraints in Fig. 1.1b, DocTer automatically generates the bug-triggering input in Fig. 1.1c. The four pairs of square brackets indicate that the parameter `grid` is four-dimensional (4-D). The two elements in `grid`, i.e., “(2.3e+38, 0)”, are the indices to specify a pixel in a given image (the image is another parameter of `grid_sample` not shown for simplicity). The large index value (2.3e+38) causes the computation to produce `NaN` (not a number), which leads to an invalid array access, resulting in a segmentation fault. This bug is only triggered in the `padding_mode = "reflection"` mode with a large index value in the `grid`’s tensor. A fuzzing technique that randomly generates inputs for this API fails to generate any input to trigger this bug.

In addition to valid inputs, DocTer generates invalid inputs that violate the constraints to test the input validity checking code of API functions. Despite invalid inputs, DL API functions should not crash, because anyone can invoke API functions, and may make mistakes due to carelessness, ignorance, or malice. Thus, one expects API functions to report the invalid input (e.g., by throwing an exception or printing an error message) instead of crashing. 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*”. Thus, we also consider that serious failures such as crashes caused by invalid inputs indicate bugs.

Since incorrect API documentation provides false information about APIs, which often misleads developers to introduce bugs in code [102], it is important to detect bugs in API documents as well. Different from prior work [102, 105] that detects inconsistencies between documents/comments and code, DocTer detects inconsistencies within documents. For example, if the shape of a parameter is dependent on another parameter, which is missing in the API document, the document is inconsistent, indicating a *documentation bug*.

1.2 Contributions

In this work, we make the following contributions:

- A technique that automatically extracts 17,919 constraints from API documentation with the focus on four categories of input properties in DL APIs: *dtype*, *structure*, *shape*, and *valid values* for 2,273 API functions across the three widely-used DL libraries, TensorFlow, PyTorch, and MXNet [23]. The constraint extraction accuracy is 82.2–90.5%.
- Use of sequential pattern mining to help identify patterns from API documentation for constraint extraction.
- A new fuzz-testing technique capable of generating multi-dimensional array object inputs (i.e., tensors) as well as other general inputs, guided by the four categories of constraints above.
- A prototype *DocTer* that detects 46 bugs in the three libraries, while a baseline fuzzer that has no knowledge of constraints detects 19 bugs only. *DocTer* detected all the bugs that the baseline detected, i.e., 27 (46 – 19) bugs are found by *DocTer* but not the baseline. Among the 46 bugs, 33 are previously unknown bugs, 26 of which have already been fixed (13) or confirmed (13) by the developers after we report them. In addition, *DocTer* detects 37 documentation bugs, 25 of which have already been fixed (10) or confirmed (15) after we report them.

Chapter 2

Approach

2.1 Challenges and overview

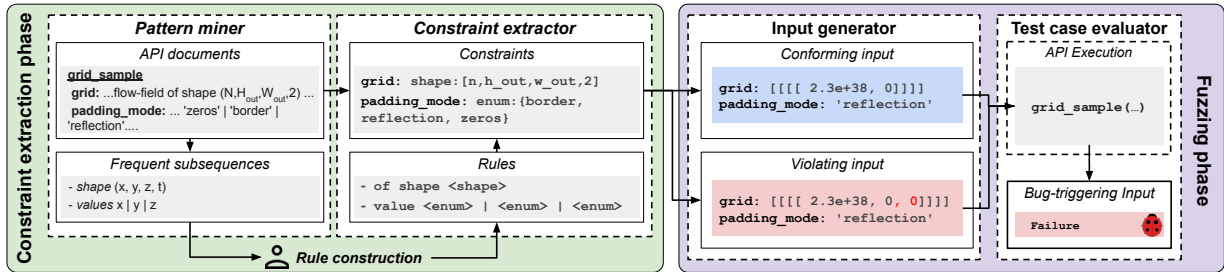


Figure 2.1: Overview of DocTer

Fig. 2.1 shows the overview of DocTer using an example of the PyTorch function `grid_sample` (whose document is in Fig. 1.1a). The *constraint extraction* phase takes API documents and extracts constraints for each input parameter. The *fuzzing* phase takes these constraints, generates test inputs either conforming or violating the constraints, and evaluates the generated inputs to return bug-triggering inputs.

Each component comes with its own challenges. A major challenge of the constraint extraction phase is analyzing free-form API documentation written in a natural language [118, 105, 18, 124, 37, 29, 130, 84]. We leverage *sequential pattern mining* [42, 46] (SPM) to collect frequently occurring patterns and manually transform them into constraint extraction rules (or *rules* for short) to automatically extract constraints from API

documents. Our semi-automated process reduces the manual effort required to discover useful patterns in a large number of documents. The constraint extraction works together with the rules, matches certain keywords relevant to the properties of a parameter such as *dtype* and *shape*, and outputs the parameter’s constraints.

One challenge for the fuzzing phase is satisfying the constraint dependencies (i.e., the relation between the properties of different parameters). Generating inputs that follow such constraints requires DocTer to determine the parameters generation order correctly to ensure that the generated values of earlier parameters do not break the constraint dependency with later parameters. We address this by generating the values for parameters using the topological order of the dependency graph. This graph is a directed acyclic graph that represents the parameter dependencies, where a node represents a parameter and a directed edge represents a dependency.

The semi-automatic constraint extraction phase consists of two components, *Pattern miner* and *Constraint extractor*. The pattern miner automatically finds frequent subsequences from the sentences in the API documents. These subsequences are manually verified and transformed into rules (up to 15 hours per project). The constraint extractor takes these rules and automatically extracts a set of constraints for each input parameter.

For example, in PyTorch documents (e.g., Fig. 1.1a), subsequences “*shape (x, ...)*” often specify the *shape* of a corresponding parameter. Using these frequent subsequences, we create a rule that extracts the *shape* of parameters using regular expressions (as shown in Fig. 1.1b). For `grid_sample`, the extracted *shape* constraint indicates that `grid` must be a 4-D or 5-D tensor.

During the fuzzing phase, for each DL API function, DocTer takes the extracted constraints and iteratively performs two steps: generating an input (*Input generator*) and evaluating that input (*Test case evaluator*). By either following or violating the *extracted* constraints, the input generator generates *Conforming inputs* (CIs) or *Violating inputs* (VIs), respectively. Since the extracted constraints may be incorrect or incomplete, the CIs are not always valid and the VIs are not always invalid. In this work, we consider an input a *valid input* or *invalid input* if it follows or violates, respectively, the ground-truth constraints, as opposed to extracted constraints. The test case evaluator uses the generated input to invoke the API function and returns the bug-triggering inputs that cause severe failures (e.g., segmentation faults).

For the function `grid_sample` (whose API document is in Fig. 1.1a), the extracted constraints (as shown in Fig. 1.1b) indicate that `grid` must be a 4-D or 5-D tensor and parameter `padding_mode` expects one of three options: "border", "reflection", "zeros". By following these constraints, the input generator creates an input (as shown in Fig. 1.1c

and explained in the Introduction). The test case evaluator executes `grid.sample` with this input and detects a segmentation fault. This PyTorch bug is previously unknown and has been fixed after we report it.

2.2 Pattern miner

Since API documentation is presented informally in natural language, manually extracting rules from the documents is expensive. For example, there are 2,334 pages of API documents and 854,900 words in TensorFlow v2.1.0. It is a daunting and tedious task for developers to manually examine such a large set of API documents to identify constraints. Following prior work [18, 105], we use rules to match potential API documents and extract relevant constraints. Different from the prior work where the authors designed rules manually, we semi-automate this process by using SPM to identify recurring patterns in API documents. In addition, our rules are designed for analyzing DL API documents, which have not been explored by existing work.

Specifically, DocTer automatically applies SPM on sentences of API documents to find *frequent subsequences*. These subsequences provide insight and templates that save manual efforts required to construct rules from scratch.

2.2.1 API document collection and preprocessing

Before applying SPM, DocTer collects API documents from DL libraries' websites. We focus on two sources in API documents for pattern mining: parameter descriptions and parameter names. Parameter descriptions in API documents often specify useful constraints such as the parameter type. Parameter names often imply additional constraints (e.g., `parameter name` implies its *dtype* to be `String`).

To extract information from API documents, we first parse the HTML documents to obtain function signatures and parameter descriptions using an HTML parsing tool [3]. Since sentences are a natural unit of organizing constraints, we split the description into sentences using regular expressions. Parameter names are extracted from API signatures. We tokenize the sentences and parameter names into words using white spaces and "_" respectively. These extracted sequences of words are fed into the SPM process as lists of items.

To improve the effectiveness of SPM, we normalize data types (e.g., `int32` and `int64`) as `D_TYPE` and structure types (e.g., `array`, `list`, and `tuple`) as `D_STRUCTURE`. This way,

references to different data and structure types could be grouped into the same frequent sequential pattern, reducing the number of frequent subsequences for less manual inspection effort. Fortunately, each library provides a list of supported data types. There are 23, 13, and 13 *dtypes* in these lists for TensorFlow, PyTorch, and MXNet, respectively. Since the lists use the exact data types (e.g., `np.int32`), we add informal variations (e.g., “int32” and “integer”) to match the format of API documents. We also add a common *dtype* that is missing such as `String`. In total, we use 30, 18, and 18 type phrases for TensorFlow, PyTorch, and MXNet, respectively. We manually collected 7 structure types that are shared by all three libraries.

2.2.2 Sequential pattern mining

It is hard to manually discover rules, because API documents use many different ways to specify the same content. For example, one common way to specify the *dtype* of a parameter is “*must be one of the following types: D_TYPE₁, ... D_TYPE_n*”, which occurs 175 times in all API documents. Another way is “*tensor type D_TYPE*” (occurs 135 times). In addition, the sentence “*If set to `true`,...*” implies that the *dtype* of the parameter is `boolean` (found by the pattern “*set true/false*” which occurs 110 times). These examples show that it would be difficult, tedious, and error-prone for developers to manually design extraction rules.

Therefore, DocTer uses SPM to automatically find frequent subsequences from the sentences in the API documents. SPM discovers frequent subsequences within a sequence dataset [69]. For example, an input sequence dataset D can consist of a set of sequences: $D = \{ \langle \underline{a}, b, c, \underline{d} \rangle, \langle \underline{a}, c, \underline{d}, b \rangle, \langle \underline{a}, e, c, \underline{d}, f \rangle, \langle b, \underline{a}, e, \underline{d} \rangle \}$. If a subsequence $\langle a, d \rangle$ appears four times in the dataset, its `support` and `length` are 4 and 2, respectively. From dataset D , an SPM algorithm finds all frequent subsequences that occur at least `min_support` times and have a length of at least `min_len` [14] efficiently, while a naive approach that counts the frequencies of all possible patterns does not scale. DocTer uses PrefixSpan SPM [46], for its efficient processing. We select the same `min_support` and `min_length` thresholds for all three evaluated DL libraries to demonstrate the generality.

2.3 Rule construction

We manually categorize and convert the frequent subsequences, mined by the pattern miner, into four categories (i.e., *structure*, *dtype*, *shape*, and *valid value*) of rules. We

Table 2.1: Rule examples and the extracted constraints

Category	No.	Examples of extraction rules	Examples of sentences from API documents	Examples of extracted constraints
<i>structure</i>	1	<code><structure></code> (list/tuple/...) of <code><dtype></code>	n: A list of integer.	n: <code>structure={list(int)}</code>
	2	<code><structure></code> (dict/dictionary) of <code><dtype₁></code> to <code><dtype₂></code>	features: Dict of string to 'Tensor'.	features: <code>structure={dict(string:tensor)}</code>
<i>dtype</i>	3	of/with type <code><dtype></code>	audio: A 'Tensor' of type 'float32'.	input: <code>dtype={float32},structure={tensor}</code>
	4	<code><ndim></code> -d/dimensional <code><dtype></code> tensor	mask: K-D boolean tensor.	mask: <code>dtype={boolean},ndim={k},structure={tensor}</code>
	5	must have the same type/dtype as <code><dependency></code>	imag: Must have the same type as 'real'.	imag: <code>dtype={&real.dtype}</code>
<i>shape</i>	6	<code><ndim></code> -d/dimension tensor	logits: 2-D Tensor..	logits: <code>ndim={2},structure={tensor}</code>
	7	with/of the same shape as <code><dependency></code> ,	target: A tensor with the same shape as 'output'.	target: <code>shape={&output.shape}</code>
	8	of/with shape <code><shape></code>	weights: ...of shape '[num_classes, dim]'.	weights: <code>shape={ [num_classes,dim]}</code>
	9	tensor of length <code><shape></code>	rates: A 1-D Tensor of length 4.	rates: <code>shape={ [4]}</code>
<i>valid value</i>	10	only <code><value₁></code> , ..., <code><value_n></code> are supported	data.format: A string. Only "NWC" and "NCW" are supported.	data.format: <code>dtype={string},enum={"NWC","NCW"}</code>
	11	non-negative <code><dtype></code>	num_columns: ... non-negative integer...	num_columns: <code>range={ [0,inf]},dtype={int}</code>
	12	must be in the range <code><valid range></code> .	axis: Must be in the range '[-rank(input), rank(input))'	axis: <code>range={ [-&input.ndim,&input.ndim]}</code>

focus on these four categories because they represent the most common properties of input parameters of API functions in major DL libraries. With these four categories, DocTer is able to extract constraints from almost all (97.8%) of the collected API functions in TensorFlow, PyTorch, and MXNet. The four categories are:

- *structure*: the type of data structure that stores a collection of values for the input parameter, such as list, tuple, n-dimensional array (i.e., tensor), etc.
- *dtype*: the data type such as `int`, `float`, `boolean`, `String`, etc., of the parameter or the elements of *structure*.
- *shape*: the shape or number of dimensions of the input parameter. For example, in row 8 of Table 2.1, `weights` has a shape of `[num_classes, dim]` (i.e., it is a 2-dimensional array with the sizes of the first and the second dimension being `num_classes` and `dim`, respectively).
- *valid value*: a set of enumerated values (e.g., parameter `padding` can only take three possible values: "zeros", "border", and "reflection") or the valid range of a numeric parameter (e.g., a float between 0 and 1).

Table 2.1 shows examples of rules (column “Examples of extraction rules”) and examples of matched sentences (column “Examples of sentences from API documents”). For example, the first rule “`<structure>` (list/tuple/...) of `<dtype>`” is used to extract the structure of a parameter, which could be applied to “n: A list of integer” (where n is the parameter name).

We make several reasonable assumptions when constructing the rules. For example, a parameter is assumed to be a 0-dimensional `float` between 0 to 1 inclusive if the document states it is a “*probability of ...*”. In addition, the `dtype` of a parameter’s default value is considered to be one of the valid *dtype* for that parameter.

The frequent subsequences extracted from parameter names represent patterns in parameter naming. We manually investigate these patterns and add *dtype* constraints using

our knowledge of DL libraries. For example, a frequent subsequence “*shape*” in a parameter name indicates that the parameter represents the shape of a tensor. Since the shape of a tensor is always a 1-D array with each array element specifying the size of a dimension, parameters with names containing “*shape*” should be a 1-D array of non-negative integers.

2.4 Constraint extractor

The constraint extractor automatically finds matching texts in the parameter descriptions and names, and extracts the relevant constraints according to the rules. For each rule in the “Examples of extraction rules” column in Table 2.1, we list one example of the sentences (column “Examples of sentences from API documents”) that can be matched. The extractor automatically generates the corresponding constraints shown in the column “Examples of extracted constraints”.

2.4.1 Constraint dependencies

The description of one parameter often refers to the *dtype*, *shape*, and *valid value* of another parameter of the same API function. In such cases, DocTer extracts constraints that involve *dependencies* among input parameters. Since most dependencies are direct (i.e., the property of one parameter is the *same* as another parameter), we do not consider less common or implied dependencies (e.g., “*compatible with*” or “*broadcastable to*”). These constraint dependencies are useful not only for generating valid inputs but also for determining the parameters’ generation order.

For *dtype* dependencies, DocTer uses extraction rules such as “*must have the same dtype as <other_parameter>*” to extract the *dtype* dependencies among parameters. For example, row 5 in Table 2.1 shows the constraint dependency `dtype: {&real.dtype}` (& symbol in front of the `<other_parameter>` indicates that it is a dependency) where `imag`’s *dtype* must be the same as `real`.

Parameters can also have *shape* dependencies. The example in row 7 of Table 2.1 shows the dependency to be the shape of parameters (i.e., the parameter `target` has the same shape as parameter `output`). The *shape* dependency could also involve sizes of the parameter’s dimensions. Example in row 8 indicates that `weights` should have shape `[num_classes, dim]` (i.e., a 2-dimensional array). The first dimension of this shape is specified by another parameter `num_classes` in the same API function. The second dimension

`dim` is a constant which is used by another parameter `inputs` with shape `[batch_size,dim]`, so parameter `weights` should have the same last dimension size as `input`.

Valid value dependencies such as *range* dependencies arise when a parameter's elements should be in a certain range using another parameter's constraints. For example, row 12 in Table 2.1 shows that the value of the parameter `axis` should be within a range determined by the rank (i.e., number of dimensions) of another parameter `input`.

2.4.2 Documentation bug detection

As discussed in the Introduction, DocTer detects inconsistencies within documents as documentation bugs. For example, the parameter names in the description should match the names specified in the function signature. If they mismatch, there is an inconsistency in the document. Since DocTer is capable of analyzing constraint dependencies, DocTer also detects dependency inconsistencies in documents. For example, in the document of `tf.keras.backend.moving_average_update`, the description for parameter `value` is “A tensor with the same shape as `variable`,...”. However, parameter `variable` is not documented, which indicates a documentation bug of unclear constraint dependency. DocTer detects this document inconsistency, which has been fixed after we report it.

2.5 Fuzzing process

For each API function, DocTer performs fuzzing by iteratively generating an input and evaluating that input. It generates two types of input: *conforming input* (CI) and *violating input* (VI). The conforming inputs are designed to test the core functionality of the API function while the violating inputs are designed to test the API functions' 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 `conformRatio`.

2.5.1 Input generator

The input generator generates one Python input for each fuzzing iteration. Given a set of extracted constraints, DocTer generates a value for each parameter following the generation

order (determined by the constraint dependencies as described in Section 2.1). For a conforming input, all generated arguments satisfy extracted constraints. Specifically, for each parameter, the input generator picks a value that conforms to the relevant *dtype*, *shape*, *valid value*, and *structure* constraints. If concrete values are specified in the constraints, the input generator uses those values. Otherwise, it chooses a *dtype* from the list of *dtypes* specified in the parameter constraints and creates a *shape* following the constraints. If the constraints do not specify a list of valid *dtypes*, DocTer selects a *dtype* from a default list of all possible *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 based on the given *dtype* and *shape*. If there is a valid range constraint (e.g., must be a value between 0 and 1), the input generator only generates values within the specified range. Finally, the *structure* constraints are checked and satisfied. Specifically, if the generated value is a 1-dimensional array and the constraints specify the parameter should be an explicit *structure* (e.g., a tuple or a list), the input generator converts the generated value accordingly.

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

2.5.2 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 for both conforming and violating inputs. More precisely, DocTer returns those inputs causing a segmentation fault, floating-point exception, bad memory allocation, and hang in C++ core-level as bug-triggering inputs. Core-level C++ signals indicate severe problems because DL libraries use C++ code to handle computationally-intensive tasks. DocTer finds many abort signals that occurred due to C++ assertion failures. We do not view this behavior as buggy, but as bad coding practice because users mostly invoke the API functions in Python-level. Therefore, DocTer does not report C++ assertion failures as bug-triggering inputs.

Chapter 3

Experiments and Results

3.1 Data collection

We choose three popular DL libraries (TensorFlow 2.1.0, PyTorch 1.5.0, and MXNet 1.6.0) as testing subjects. There are 144,541–854,900 words in the API documents among the subjects. We collect documents for 949, 415, and 959 relevant API functions in the three libraries, respectively. A function is considered irrelevant if it (1) is deprecated, (2) is from an old version, (3) is a class constructor, (4) has no input argument, (5) has API document without a “Parameter” description section, or (6) has a detected documentation bug (e.g., format and mismatch issues).

3.2 Constraint extraction

We apply PrefixSpan [46] SPM on parameter names and parameter descriptions with NLTK [17] English stop words excluded. For parameter names, we set *min_len* = 1 to target short patterns (e.g., “*shape*” or “*name*”). These patterns imply the *dtype* of the parameter (e.g., `String` for “*name*”). For parameter descriptions, we set *min_len* = 2 to catch more descriptive patterns (e.g., “*positive D_TYPE*” or “*tensor of length ...*”). We set *min_support* = 5 for both parameter names and descriptions so that we can cover most of subsequences with reasonable manual inspection effort. Depending on resources available, one can decrease *min_support* to find more rules with increased manual effort or increase this number for a quicker manual inspection at the risk of missing rules.

3.3 Fuzzing

Since all three libraries heavily depend on `numpy` package, DocTer uses `numpy` array format to generate input values. In this way, the subject library conveniently adapts the generated input. For each generated input, once a timeout of 10 seconds is reached, DocTer terminates the evaluation process and moves on to the next iteration.

3.4 Constraint extraction results

3.4.1 Approach

We apply DocTer to extract constraints in our subjects and study the number and quality of constraints. We randomly sample 5% of input parameters of API functions from each subject, which results in a total of 534 parameters. For each sampled parameter, we extract relevant constraints manually to build the ground truth that we use to evaluate the accuracy of our semi-automated constraint extraction. We extract constraints of the four categories: *dtype*, *structure*, *shape*, and *valid value*, because other constraints are out of scope by design of our approach. Chapter 4 discusses other categories which remain as future work. The ground truth constraints of the sampled parameter are extracted independently by two authors who have reached 91% agreement. For any disagreement, we reach a consensus after discussing it with a third author. The extracted constraints by DocTer are compared against manually extracted ground truth constraints for evaluation.

To estimate the quality of extracted rules, we calculate the *accuracy* of the constraint extraction for the sampled parameters (i.e., the percentage of correctly analyzed parameters over the total number of sampled parameters). We use a very strict definition of accuracy—a parameter is correctly analyzed if DocTer accurately extracts all constraints of the four categories for this parameter. For example, parameter `size` of `tf.slice` can be either `int32` or `int64`. The extracted *dtype* constraint `size:dtype={int32,int64}` is deemed correct, while `size:dtype={int32}` is considered incorrect. If a parameter’s document contains no constraints of the four categories, the parameter is excluded from this accuracy and subsequent 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 no-constraint parameters. Among the sampled parameters, the numbers of no-constraint parameters are 10 (6.0%), 8 (11.3%), and 49 (16.6%) for TensorFlow, PyTorch, and MXNet, respectively (details in Extraction result section below).

Table 3.1: Quality of constraint extraction

	TensorFlow	PyTorch	MXNet	Total/Avg
# APIs with constr. extracted	915	404	954	2,273
# constr. extracted	6,729	2,201	8,989	17,919
# constr. per API: <i>Avg (Min-Max)</i>	7.4 (1-44)	5.4 (1-34)	9.4 (1-143)	7.9 (1-74)
# examined param.	167	71	296	534
# examined param. with constr.	157	63	247	467
# examined constr.	341	108	442	891
Accuracy (%)	82.2±5.8	90.5±7.1	90.3±3.6	87.7
Precision/Recall for <i>structure</i> (%)	97.4/96.2	96.8/96.8	98.3/99.1	97.5/97.4
Precision/Recall for <i>dtype</i> (%)	93.4/90.4	97.4/97.4	96.9/94.5	95.9/94.1
Precision/Recall for <i>shape</i> (%)	92.3/93.2	90.9/93.8	95.9/90.6	93.0/92.5
Precision/Recall for <i>valid value</i> (%)	90.3/82.4	100/83.3	94.3/97.1	94.9/87.6
Precision/Recall for All (%)	93.7/91.8	95.4/95.4	96.8/94.8	95.3/94.0

To show the quality of constraints extracted for each parameter, we compute the precision and recall 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.

3.4.2 Extraction results

Table 3.1 shows the quality of extraction. In total, DocTer extracts 17,919 constraints from the three libraries (row “# *constr. extracted*”). Specifically, DocTer extracts a total of 3,773 and 533 frequent subsequences from 18,754 sentences in the parameter description and 10,772 parameter names, respectively. We exclude 56.7% subsequences that either contains no constraints (49.9%), e.g., functionality description of the API functions, or describes constraints out of the scope of DocTer (6.8%), e.g., complex constraints (discussed in Chapter 4). For example, we exclude subsequence “*cuda operator*” because some sentences that contain this subsequence, e.g., “*Do not select CUDA operator, if available.*”, do not contain the constraints in our scope. After removing irrelevant subsequences, we keep 1,757 frequent subsequences from parameter descriptions and 108 from parameter names. We then group the relevant subsequences together and obtain in total 239 con-

straint extraction rules, where each rule is merged from 7.8 subsequences on average. For example, we group “*the input array*” and “*the output array*” together to get the rule “*the input/output array*” which indicates `array` as one of the valid *structure*. Using these rules, DocTer extracts on average 7.4 constraints per API for TensorFlow, 5.4 for PyTorch, and 9.4 for MXNet (row *# constr. per API: Avg (Min-Max)* Table 3.1). Overall, DocTer is able to extract constraints from 96.4%, 97.3%, and 99.5% of relevant APIs (details in Chapter 3) for TensorFlow, PyTorch, and MXNet, respectively.

For each subject, Table 3.1 shows the number of manually examined parameters (row *# examined param.*), the number of manually examined constraints (row *# examined constr.*), and the number of examined parameters with at least one constraint (row *# examined param. with constr.*). The *Total/Avg* column shows the total number of examined parameters and constraints as well as the average accuracy, precision, and recall. The confidence intervals of accuracy are computed with 95% confidence level.

Overall, DocTer achieves a high accuracy of constraints extraction (87.7%) across all three subjects. DocTer is the most accurate in extracting constraints for PyTorch and MXNet with a high accuracy over 90%, precision over 95%, and recall over 94%. DocTer is less accurate when extracting constraints for TensorFlow with a good but lower accuracy (82.2±5.8%). The reason is that sentences in TensorFlow’s API documents are longer and more free-form compared to other subjects. Hence, it is much harder for DocTer to extract a complete set of constraints for each parameter. However, we can still extract thousands of correct constraints for TensorFlow, which helps generate valid inputs and detect more bugs.

DocTer achieves high precision and recall (over 90%) for *structure*, *dtype*, and *shape* constraints. For *valid value* constraints, we achieved lower recall for TensorFlow and PyTorch because we miss some uncommon patterns in the mining stage (due to high threshold `min_support = 5`). Reducing the threshold of our sequential pattern mining could help us extract more rules, which remains as future work.

3.4.3 Generality of rules

Since all three subjects are DL libraries, one may wonder the amount of similarity in their API documents and extraction rules. Table 3.2 shows the number of rules (column *Rules*) and the corresponding extracted constraints (column *Constraints*) for each category of each library. The last column shows the number of rules that are shared among the libraries. The *overlapping ratio* is the number of shared rules divided by the minimum number of rules in the three libraries. For example, 12 of the *dtype* rules are the same across the

Table 3.2: Rule overlap across the three libraries

Category	TensorFlow		PyTorch		MXNet		Rule Overlap
	Rules	Constraints	Rules	Constraints	Rules	Constraints	Rules (Ratio)
<i>dtype</i>	41	2,662	38	855	47	3,208	12 (31.6%)
<i>structure</i>	13	1,245	12	643	16	2,454	4 (33.3%)
<i>shape</i>	16	2,235	13	623	11	2,584	2 (18.2%)
<i>valid value</i>	15	587	5	80	11	743	0 (0.0%)
Total	85	6,729	68	2,201	85	8,989	18 (26.5%)

libraries, resulting in an overlapping ratio of 31.6%, which is $\frac{12}{\min(41,38,47)}$, since at most only $\min(41, 38, 47)$ number of rules can be shared across the libraries.

Overall, 18 rules are shared among the three libraries with an overlapping ratio of 26.5%. On one hand, this result indicates the generality of the extracted rules and suggests that a significant portion of the extracted rules can be reused in extracting constraints for new libraries. For example, “*<ndim>-d/dimension tensor*”, one of the shared rules, captures a common way among all three libraries to describe the number of dimensions of an input **tensor**. Phrases such as “*a list of `strings`*” and “*tuple of floats*” are used in all three libraries to describe the *structure* of an input parameter which can be matched with a shared rule “*<structure> of <dtype>*”.

On the other hand, the 18 shared rules only account for a small portion of the total number of rules for all three libraries. The rest of the rules are unique to one or two libraries. For example, the *valid value* rule “*<enum₁>, <enum₂> ... are supported*” is unique to TensorFlow and MXNet. To extract similar *valid value* constraints in PyTorch, DocTer uses rules such as “*can only be <enum₁>, <enum₂> ...*”, e.g., “*`signal_ndim` can only be 1, 2 or 3.*”, which is unique to PyTorch. Given the diversity of the rules, the proposed semi-automatic process using sequential pattern mining is essential to reduce manual effort in applying DocTer to new libraries.

3.5 Bug detection results

3.5.1 Approach

We evaluate DocTer’s effectiveness in detecting bugs in both API documents and library code, as both hurt software reliability now or later [102]. For the documentation bugs,

Table 3.3: Number of **verified new** / **new** / **all** bugs found by DocTer and the baseline

Approach		TensorFlow	PyTorch	MXNet	Total
Baseline		5 / 8 / 14	4 / 4 / 4	1 / 1 / 1	10 / 13 / 19
DocTer	All	8 / 15 / 25	10 / 10 / 11	8 / 8 / 10	26 / 33 / 46
	CI	6 / 13 / 21	6 / 6 / 6	7 / 7 / 9	19 / 26 / 36
	VI	8 / 13 / 21	6 / 6 / 7	8 / 8 / 9	22 / 27 / 37

we use DocTer to detect inconsistencies within API documents when analyzing them. For library code bugs, we use DocTer to test API functions that have at least one constraint extracted. Table 3.1 shows the numbers of these API functions (row # *APIs with extracted constr.*). For each API function, with the `conformRatio` set to 50%, DocTer generates 1,000 test inputs (500 conforming inputs and 500 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. Since there are no state-of-the-art API fuzzers for DL libraries, we implement an unguided fuzzer as the *baseline* for comparison. The baseline generates random inputs for all 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 argument should be a tensor. Without this conversion, non-tensor input arguments are trivially rejected by PyTorch and MXNet, thus very ineffective in exercising the code in depth.

3.5.2 Bugs in library code

Table 3.3 presents the number of *verified new* / *new* / *all* bugs found by DocTer and the baseline. 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 have reported.

DocTer detects 46 bugs including 33 new bugs, 26 of which have been verified by the developers (13 fixed and 13 confirmed). On the other hand, the baseline detects only 19 bugs with 13 new bugs. DocTer detects all 27 (46 – 19) bugs that the baseline cannot, while missing no bug found by the baseline. The unverified new bugs are reproducible and waiting for developers’ response. None of the newly reported bugs has received a ‘won’t fix’ response from the developers, which suggests that our bug-triggering inputs are not

false alarms and trigger the real bugs. DocTer has also uncovered 13 (*all bugs* – *new bugs*, 46 – 33) known bugs that have already been fixed in the nightly versions.

Our fuzzer takes the automatically extracted constraints without any manual examination. The few incorrectly extracted constraints could potentially hurt the fuzzer’s effectiveness, but in practice, the impact is small as shown by our strong bug detection results. Alternatively, one can manually examine all extracted constraints first, if they would like to trade manual effort of constraint verification for higher bug detection effectiveness. It is possible that documents themselves are incorrect, causing incorrect constraints to be extracted, leading the fuzzer to produce false alarms. Since we focus on severe bugs such as crashes, all detected bugs are in the library code, 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.*”

3.5.3 Conforming and violating inputs

As discussed earlier, DocTer generates both conforming inputs (CIs) and violating inputs (VIs). Table 3.3 presents the breakdown of the bug detection result for CIs and VIs with `conformRatio = 50%`. The results show that the CIs alone (with half of the test cases of the baseline) find more bugs (36 bugs) than the baseline (19 bugs), and the VIs alone (with half of the test cases) also find more bugs (37) than the baseline. We manually verified the generated CIs and VIs: out of 36 CI bugs we found, 27 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 37 VI bugs we found, all of them are caused by invalid inputs violating the ground truth constraints.

It is widely known and expected that CIs find more bugs than the baseline, because most CIs are expected to be valid inputs, exploring deeper to find hard-to-detect bugs, while the baseline mostly generates invalid inputs which get stuck at the input validity checking code. The reason that VIs find more bugs than the baseline is that the input validity checking code is often complex. The baseline mostly generates inputs that violate all parameter constraints, thus stops at the first batch of checks of input validity. On the other hand, DocTer violates the constraints of one parameter only for each VI, thus can pass the checking of all other parameters, going much deeper than the inputs generated by the baseline.

By comparing the “All” row with the “CI” and “VI” rows, we can see that many bugs are detected by both CIs and VIs. This is 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 also likely to be triggered by a CI. However, both CIs and VIs detect unique bugs that cannot be detected by the other, thus both types of inputs are effective in detecting bugs.

While the optimal ratio between the CIs and VIs may depend on the type of applications and maturity of code (for example, there may be more bugs in the input validity checking and exception handling code in mature projects as the norm cases have been tested more thoroughly), we study how this ratio (`conformRatio`) affects the testing effectiveness on the three DL libraries. We perform our study by varying `conformRatio` between 0% to 100% with a 10% increment. Overall, the numbers of bugs detected using ratio between 20% to 90% differ by at most three bugs which indicates that DocTer is insensitive to this ratio. Thus, we use `conformRatio = 50%` as the default ratio to be more general.

Although the optimal ratio of conforming and violating constraints may change for different projects, it is crucial to know the constraints. We have observed that without the constraints, a baseline is much worse than the results from any of the ratio setups. Table 3.3 shows that a *key contribution of our work is the ability to extract constraints from documents*. Regardless of what inputs (valid or invalid) to focus on, one cannot make such a choice without knowing the definition of valid inputs for an API. DocTer enables this choice by extracting input constraints from DL API documents, and use these constraints to guide fuzzing to find more bugs.

3.5.4 Bugs in API documents

In addition to the bugs in the library code, DocTer also detects documentation bugs during the constraint extraction phase. DocTer can detect three types of documentation bugs: (1) formatting bugs (e.g., indentation issue); (2) signature-description mismatch (e.g., the description refers to a parameter that is not specified in the API signature); and (3) unclear constraint dependency (e.g., as discussed in Section 2.4, some properties of one parameter depend on those of a missing parameter). We detect the first two types during the API documents collection and the third when extracting constraints with rules.

DocTer detects 37 previously unknown documentation bugs for 53 API functions in three libraries (including 9 formatting bugs, 25 signature-description mismatches, and 3 unclear constraint dependencies). We report all 37 documentation bugs, 25 of which have been fixed or confirmed by the developers (including all 3 unclear dependencies). The result suggests that, in addition to code bugs, DocTer is effective in detecting documentation bugs that developers care to fix.

3.6 Bug examples

We present three examples of bugs detected by DocTer that the baseline fails to detect. All of them have been fixed by developers after we report them. **Bug 1** is the previously unknown bug in PyTorch API `grid_sample` discussed in the Introduction (Fig. 1.1).

3.6.1 Bug 2

DocTer detects a floating-point exception bug in API `mxnet.ndarray.InstanceNorm`. The bug involves three parameters, `data`, `gamma` and `beta`, which have *shape* dependency on each other: `gamma` and `beta` must have the same length which should be equal to the size of the second dimension of `data`. When at least one of the sizes of the first two dimensions of `data` is 0, e.g., when it is of shape such as $(0, 1, \dots)$ or $(1, 0, \dots)$, a division by zero exception occurs. The fix adds code to check zero before the division.

3.6.2 Bug 3

DocTer detects a segmentation fault in API `torch.cholesky_solve`. This function requires two input tensors to be at least 2-D, and their second to last dimensions should match. The API function takes these two tensors and performs broadcasting (i.e., making their shapes to be compatible for the arithmetic operations). However, the API execution fails to check the shapes for compatibility and attempts to perform broadcasting with the incompatible tensors. This causes a segmentation fault due to an invalid array access. The fix calls a different function that properly checks the shape compatibility before broadcasting.

3.6.3 Discussion

The baseline approach fails to generate valid inputs and cannot trigger any of these bugs. Unless generated inputs conform with the strict constraints, the input validity checking of the API functions rejects the inputs by throwing exceptions. Therefore, the inputs generated by the baseline do not exercise the core functionality, which contains bugs. The inputs generated by DocTer, however, pass the validity check and find the bugs by exploring deeper into the program.

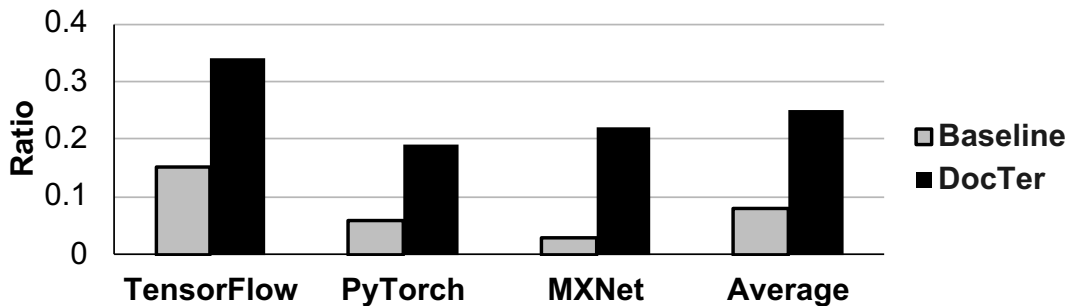


Figure 3.1: Ratio of passing inputs

3.7 Valid-input generation results

3.7.1 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 what percentage of generated CIs are valid inputs. For a fair comparison, we compare 1,000 CIs with 1,000 baseline inputs. The ratio of valid inputs is computed out of 1,000 total inputs generated for each API function.

Since manually examining the validity of all inputs for all API functions under test is impractical, we approximate the number of valid inputs by counting the number of passing inputs whose executions terminates without any exceptions or crashes from both Python and C++ level. Since the validity checking of mature projects (e.g., our subjects) is generally reliable, the passing inputs that have not been rejected by the API functions are likely valid inputs.

3.7.2 Results

Fig. 3.1 presents the ratio of passing inputs for each subject and the average. DocTer outperforms the baseline approach by generating more than three times of passing inputs that the baseline generates on average. The results suggest that DocTer is likely much more effective in generating valid input than the baseline to test the core functionality code to detect more bugs.

Although DocTer outperforms the baseline for all DL libraries, the ratio of passing inputs is relatively low at 25% on average. This is because API documents are often

incomplete. For example, many API documents specify only `ndim` of a tensor and often do not specify the size of each dimension. Thus, even though our input conforms to all constraints given in the document, the source code throws an exception to complain that the input is invalid. Therefore, the effectiveness of generating valid inputs relies on the quality of documents. We hope that DocTer can convince developers to write more complete API documents after learning that API documents can help them find bugs.

Chapter 4

Threats to Validity

4.1 Practicality of generated inputs

Many inputs DocTer generates are boundary inputs that are too large or too small because for each dtype, DocTer draws values from the uniformly distributed value range. However, these corner case inputs are beneficial to improve the robustness of standalone API functions by detecting 26 new bugs that developers have already fixed or confirmed after we report them.

4.2 Complex constraints

This work does not use some complex constraints: constraints that require (1) a class object, (2) a pointer to a function, (3) a nested structure, and (4) indirect dependency with the constraints of another parameter (discussed in Section 2.4). However, these complex constraints are uncommon in DL libraries (appeared in only 6.4% of our sampled parameters), thus excluding them should not affect the effectiveness of DocTer much.

4.3 Manual rule construction

The extraction rules need to be manually constructed from the frequent subsequences. For the three libraries in this work, it takes one person up to 15 hours per library. The process

is also only a one-time cost. We hope DocTer’s results can convince developers to write documentation in more consistent formats/expression patterns, so tools such as DocTer can more easily extract useful information to help detect bugs in the libraries.

4.4 Python test inputs

One may argue that DL libraries’ core computations are in C++, so directly testing C++ code is more appropriate. However, since Python APIs are the most popular for DL, testing them is more aligned to the popular use case. DocTer test the Python APIs which invoke the computations in C++, so DocTer can find bugs in both Python and C++ code. DocTer found one bug in Python, which hangs with an infinite loop. Since DocTer focus on severe failures (section 2.5.2), the rest of the 45 bugs DocTer found are in C++ code. Other indicator(s) is need to find more bugs in Python.

Chapter 5

Related Work

5.1 Testing DL libraries

Among all DL library testing techniques from research and industry, DocTer is the first that extracts input constraints semi-automatically to guide testing. The handful of DL libraries testing techniques focus on addressing the test oracle challenge. They leverage differential testing [90, 112, 116, 99, 28, 44] or oracle approximation [74, 129] to obtain oracles. DocTer uses only crashes as an indicator of unexpected behaviors, 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) [28, 56], numerical [44, 28] (e.g., returns `NaN/Inf`), decreased accuracy [28], and performance [108]. To do this, they leverage user-specified template to generate numerical values from the specified probability distribution [28], modeling of library semantics with respect to tensor shape [56], different combinations of structure, parameters, weights, and data input of a DNN model [44], and performance clustering [108]. DocTer can find general bugs that lead to severe crashes by generating input arguments of API functions based on the constraints specified in the documentation.

TensorFlow developers use OSS-Fuzz [7] along with libFuzzer [5] to test only 19 TensorFlow’s C++ API functions. It requires developers to manually add constraints about data structures and properties for each API to reinterpret the sequence of byte-arrays returned by libFuzzer. This would take prohibitive amount of manual effort to test on the same scale of APIs that DocTer is capable to do. DocTer extracts input constraints automatically from API documents and use the constraints to generate valid inputs to test on 2,273 APIs across three libraries.

5.2 Unit test generation and fuzzing

DocTer belongs to a large body of work that generates unit tests. Random and search-based techniques [82, 25, 109, 32] generate a sequence of methods as a test case. Our work is a random testing technique that generates API function arguments (not method sequences) for DL libraries that require DL-specific constraints. Dynamic symbolic execution engines for unit testing [36, 96] generate inputs for API functions. However, these tools require heavy program analysis, causing scalability issues. In contrast, we extract constraints from API documents, which is light-weight.

DocTer is a generation-based black-box fuzzer [70]. The state-of-the-art fuzzers such as AFL [1] and libFuzzer [5] are mutation-based grey-box fuzzers guided by code coverage [70] and have been adopted to test various non-DL libraries [20, 19, 33, 58, 59, 89, 91]. However, they would not work well for DL libraries that enforce DL specific constraints for valid inputs.

5.3 Analyzing software text to detect bugs

Previous work leverages comments [102, 104, 132, 103, 18] and documents [22, 105, 130] to detect inconsistency bugs between code and its specifications. Some prior work uses software text to extract input constraints for testing by leveraging UML statecharts for test case generation [78] and translating software specifications into oracles [29, 37, 72]. Different from these techniques, DocTer uses sequential pattern mining to aid the extraction of constraints from API documents to guide input generation for testing DL libraries.

Chapter 6

Future Work

6.1 Complex constraints

DocTer currently focuses on four categories of constraints (i.e., *structure*, *dtype*, *shape*, and *valid value*), but there are more complex constraints such as requiring a parameter to be a class object or two parameters to be broadcastable. We plan to extract more complex constraints and expand DocTer’s capability in handling these complex constraints.

6.2 Input minimization

DocTer is useful to generate bug-triggering input, but the inputs generated are not always minimal. Users of DocTer may need to manually minimize the generated input to produce the minimal reproducible test input to help locate the actual bug. We could explore approaches that can help reduce the bug-triggering input to further reduce DocTer users’ workload to detect and understand the bug.

6.3 Feedback-directed fuzzing

DocTer leverages constraints as guidance and currently does not use feedback to improve its generation. We could establish metrics to dynamically improve the generation. For example, we could record the distribution of *dtype* of previously generated inputs, and direct DocTer to generate unseen *dtype* to diversify the generated inputs as much as possible.

6.4 Use address sanitizer

DocTer uses severe crashes (such as Segmentation Fault) as indicators for bugs. An improvement could be compiling the target libraries with address sanitizer, so DocTer could potentially find more bugs with address sanitizer enabled.

Chapter 7

Conclusion

We propose a fuzzing technique called DocTer that tests API functions of deep learning libraries. The technique leverages sequential pattern mining to extract input constraints from API documentation. It uses the constraints to guide the input generation for (1) valid inputs that conform to the constraints to test API function’s core functionality, and (2) invalid inputs that violate the constraints to test the input validity checking code. Our results show that DocTer achieves 82.2–90.5% accuracy for the constraint extraction. DocTer detects 46 bugs, 33 of which are previously unknown including 26 of them fixed or confirmed by developers after we report. In addition, DocTer finds 37 inconsistencies in API documents, and 25 of them have been fixed or confirmed by developers. In the future, it would be conceivable to extract other categories of DL constraints and more complex dependency-related constraints from API documents to further fuzz-test DL libraries.

References

- [1] American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl/>.
- [2] Basic Fuzzing Framework. <https://www.cert.org/vulnerability-analysis/tools/bff.cfm>.
- [3] Beautiful Soup. <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>.
- [4] Failure Observation Engine. <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=515311>.
- [5] libFuzzer – a library for coverage-guided fuzz testing. <http://llvm.org/docs/LibFuzzer.html>.
- [6] numpy. "<https://github.com/numpy/numpy>".
- [7] OSS-Fuzz. <https://github.com/google/oss-fuzz>.
- [8] scikit-learn. "<https://github.com/scikit-learn/scikit-learn>".
- [9] What is the best programming language for Machine Learning? <https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7>.
- [10] zzuf. <https://github.com/samhocevar/zzuf>.
- [11] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

- [12] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*, pages 370–384. Springer, 2009.
- [13] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, 2007.
- [14] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*, pages 3–14. IEEE, 1995.
- [15] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435, 2002.
- [16] Anand Bhattad, Min Jin Chong, Kaizhao Liang, Bo Li, and D. A. Forsyth. Unrestricted Adversarial Examples via Semantic Manipulation. In *International Conference on Learning Representations*, 2020.
- [17] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O’Reilly Media Inc, 2009.
- [18] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253, 2018.
- [19] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.
- [20] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

- [21] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.
- [22] Sandeep Chaudhary, Sebastian Fischmeister, and Lin Tan. em-SPADE: a compiler extension for checking rules extracted from processor specifications. *ACM SIGPLAN Notices*, 49(5):105–114, 2014.
- [23] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, 2015.
- [24] Ding-Ying Chiu, Yi-Hung Wu, and Arbee LP Chen. Efficient frequent sequence mining by a dynamic strategy switching algorithm. *The VLDB Journal*, 18(1):303–327, 2009.
- [25] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exp.*, 34(11):1025–1050, 2004.
- [26] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 422–431. ACM, 2005.
- [27] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, May 2008.
- [28] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing Probabilistic Programming Systems. ESEC/FSE 2018, New York, NY, USA, 2018. Association for Computing Machinery.
- [29] Michael D Ernst. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [30] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

- [31] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In Jens Knoop and Laurie J. Hendren, editors, *PLDI*, pages 234–245. ACM, 2002.
- [32] Gordon Fraser and Andrea Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb. 2013.
- [33] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 679–696. IEEE Computer Society, 2018.
- [34] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, 2020.
- [35] Fosca Giannotti, Mirco Nanni, Dino Pedreschi, and Fabio Pinelli. Mining sequences with temporal annotations. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 593–597, 2006.
- [36] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *PLDI '05*, pages 213–223, New York, NY, USA, 2005. ACM.
- [37] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 213–224, 2016.
- [38] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, Saarbrücken, Germany, July 2016.
- [39] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*, 2015.
- [40] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples, 2014.
- [41] D. Gopinath, C. S. Pasareanu, K. Wang, M. Zhang, and S. Khurshid. Symbolic Execution for Attribution and Attack Synthesis in Neural Networks. In *2019 IEEE/ACM*

41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 282–283, May 2019.

- [42] Karam Gouda, Mosab Hassaan, and Mohammed J Zaki. Prism: An effective approach for frequent sequence mining via prime-block encoding. *Journal of Computer and System Sciences*, 76(1):88–102, 2010.
- [43] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jianguang Sun. 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, ESEC/FSE 2018*, pages 739–743, New York, NY, USA, 2018. ACM.
- [44] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated Testing for Deep Learning Frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, September 2020.
- [45] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. FreeSpan: frequent pattern-projected sequential pattern mining. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 355–359, 2000.
- [46] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224. Citeseer, 2001.
- [47] Grant Haskins, Uwe Kruger, and Pingkun Yan. Deep learning in medical image registration: A survey. *Machine Vision and Applications*, 31(1):8, 2020.
- [48] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161, 2019.
- [49] Jianjun Huang, Xiangyu Zhang, and Lin Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 169–180, 2016.

- [50] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, 2014.
- [51] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of 42nd International Conference on Software Engineering, ICSE '20*. ACM, 2020.
- [52] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 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, ESEC/FSE 2019*, page 510–520, New York, NY, USA, 2019. Association for Computing Machinery.
- [53] Renáta Iváncsy and István Vajk. Frequent pattern mining in web log data. *Acta Polytechnica Hungarica*, 3(1):77–90, 2006.
- [54] Nehul Jain, Saikat Dutta, Ansuman Banerjee, Anil K. Ghosh, Lihua Xu, and Huibiao Zhu. Using Daikon to Prioritize and Group Unit Bugs. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, pages 215–233, 2013.
- [55] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I Maletic. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, 2006.
- [56] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static Analysis of Shape in TensorFlow Programs. In *ECOOP 2020*, 2020.
- [57] Marion Leleu, Christophe Rigotti, Jean-François Boulicaut, and Guillaume Euvrard. GO-SPADE: mining sequential patterns over datasets with consecutive repetitions. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 293–306. Springer, 2003.
- [58] Caroline Lemieux and Koushik Sen. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *ASE*, pages 475–485. ACM, 2018.

- [59] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *ESEC/SIGSOFT FSE*, pages 627–637. ACM, 2017.
- [60] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005.
- [61] Ming-Yen Lin, Suh-Yin Lee, et al. Fast discovery of sequential patterns through memory indexing and database partitioning. *Journal of Information Science and Engineering*, 21(1):109–128, 2005.
- [62] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 785–790, 2014.
- [63] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.
- [64] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, 2007.
- [65] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 103–116, 2007.
- [66] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov, and B. Vorster. Deep learning in the automotive industry: Applications and tools. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3759–3768, 2016.
- [67] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2018.

- [68] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, pages 120–131, New York, NY, USA, 2018. ACM.
- [69] Nizar R Mabroukeh and Christie I Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):1–41, 2010.
- [70] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2019.
- [71] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: a simple and accurate method to fool deep neural networks, 2015.
- [72] Manish Motwani and Yuriy Brun. Automatically generating precise Oracles from structured natural language specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 188–199. IEEE, 2019.
- [73] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, pages 803–813, 2014.
- [74] M. Nejadgholi and J. Yang. A study of oracle approximations in testing deep learning libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 785–796, 2019.
- [75] Yixin Nie, Haonan Chen, and Mohit Bansal. Combining Fact Extraction and Verification with Neural Semantic Matching Networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 6859–6866, 2019.
- [76] Yixin Nie, Yicheng Wang, and Mohit Bansal. Analyzing Compositionality-Sensitivity of NLI Models, 2018.
- [77] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International*

Conference on Machine Learning, volume 97 of *Proceedings of Machine Learning Research*, pages 4901–4911, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

- [78] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.
- [79] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
- [80] Piotr Ożdżyński. Text document categorization based on word frequent sequence mining. *Information Systems Architecture and Technology, Contemporary Approaches to Design and Evaluation of Information Systems, Oficyna Wydawnicza Politechniki Wrocławskiej*, pages 129–138, 2014.
- [81] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP’05*, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.
- [82] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA ’07*, pages 815–816, New York, NY, USA, 2007. ACM.
- [83] Yoann Padiou, Lin Tan, and Yuanyuan Zhou. Listening to programmers - Taxonomies and Characteristics of Comments in Operating System Code. In *Proceedings of the 31st International Conference on Software Engineering (ICSE09)*, May 2009.
- [84] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 815–825. IEEE, 2012.
- [85] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering, ICSE ’12*, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

- [86] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The Limitations of Deep Learning in Adversarial Settings, 2015.
- [87] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.
- [88] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 1–18, New York, NY, USA, 2017. ACM.
- [89] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, pages 697–710. IEEE Computer Society, 2018.
- [90] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: Cross-backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, pages 1027–1038, Piscataway, NJ, USA, 2019. IEEE Press.
- [91] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *CoRR*, abs/1811.09447, 2018.
- [92] Michael Pradel and Thomas R. Gross. Leveraging Test Generation and Specification Mining for Automated Bug Detection Without False Positives. ICSE '12, pages 288–298, Piscataway, NJ, USA, 2012. IEEE Press.
- [93] A. Ramanathan, L. L. Pullum, F. Hussain, D. Chakrabarty, and S. K. Jha. Integrating symbolic and statistical methods for testing intelligent systems: Applications to machine learning and computer vision. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 786–791, March 2016.
- [94] Noam Rathaus and Gadi Evron. *Open Source Fuzzing Tools*. Syngress Publishing, 2007.
- [95] Cindy Rubio-González and Ben Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM*

SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 73–80, 2010.

- [96] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. *ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.
- [97] Juraj Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.
- [98] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.
- [99] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. Multiple-Implementation Testing of Supervised Learning Software. In *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.
- [100] Robert Swiecki. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. URL: <https://github.com/google/honggfuzz>.
- [101] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013.
- [102] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*Icomment: Bugs or Bad Comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.
- [103] Lin Tan, Ding Yuan, and Yuanyuan Zhou. Hotcomments: How to make program comments more useful? In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS07)*, May 2007.
- [104] Lin Tan, Yuanyuan Zhou, and Yoann Padiou. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 11–20, New York, NY, USA, 2011. ACM.

- [105] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, April 2012.
- [106] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 303–314, New York, NY, USA, 2018. ACM.
- [107] Nikolai Tillmann and Peli de Halleux. Pex - White Box Test Generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008.
- [108] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries, 2020.
- [109] Paolo Tonella. Evolutionary Testing of Classes. *ISSTA '04*, pages 119–128, New York, NY, USA, 2004. ACM.
- [110] Sakshi Udeshi and Sudipta Chattopadhyay. Grammar Based Directed Testing of Machine Learning Systems. *CoRR*, abs/1902.10027, 2019.
- [111] Jonathan Uesato, Ananya Kumar, Csaba Szepesvari, Tom Erez, Avraham Ruderman, Keith Anderson, Krishnamurthy, Dvijotham, Nicolas Heess, and Pushmeet Kohli. Rigorous Agent Evaluation: An Adversarial Approach to Uncover Catastrophic Failures, 2018.
- [112] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. Discovering discrepancies in numerical libraries. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis, ISSTA 2020*, pages 488–501, 07 2020.
- [113] Haohan Wang, Da Sun, and Eric P. Xing. 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, 2018.
- [114] Jianyong Wang, Jiawei Han, and Chun Li. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1042–1056, 2007.

- [115] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug Detection with N-gram Language Models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 708–719, New York, NY, USA, 2016. ACM.
- [116] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 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*, 2020.
- [117] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 408–426, Cham, 2018. Springer International Publishing.
- [118] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 620–631. IEEE, 2015.
- [119] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, 2006.
- [120] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 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, ISSTA 2019*, pages 146–157, New York, NY, USA, 2019. ACM.
- [121] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM international conference on data mining*, pages 166–177. SIAM, 2003.
- [122] Jinqiu Yang, Erik Wittern, Annie TT Ying, Julian Dolby, and Lin Tan. Towards extracting web api specifications from documentation. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 454–464. IEEE, 2018.
- [123] Mohammed J Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1-2):31–60, 2001.

- [124] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 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*, 2020.
- [125] Juan Zhai, Xiangzhe Xu, Yu Shi, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. CPC: automatically classifying and propagating natural language comments via program analysis. 2019.
- [126] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM, July 2020.
- [127] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on TensorFlow program bugs. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis, ISSTA 2018*, pages 129–140, 07 2018.
- [128] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. Detecting Numerical Bugs in Neural Network Architectures. 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*, 2020.
- [129] W. Zheng, W. Wang, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie. Testing untestable neural machine translation: An industrial case. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 314–315, 2019.
- [130] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE, 2009.
- [131] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. *CoRR*, abs/1812.10812, 2018.
- [132] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs documentation and code to detect directive defects. In

2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 27–37. IEEE, 2017.

- [133] Zhi Quan Zhou and Liqun Sun. Metamorphic Testing of Driverless Cars. *Commun. ACM*, 62(3):61–67, February 2019.