Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source

Anjiang Wei*
Stanford University
anjiang@stanford.edu

Yinlin Deng
University of Illinois at Urbana-Champaign
yinlind2@illinois.edu

Chenyuan Yang*
Nanjing University
cy1yang@outlook.com

Lingming Zhang
University of Illinois at Urbana-Champaign
lingming@illinois.edu

ABSTRACT

Deep learning (DL) systems can make our life much easier, and thus are gaining more and more attention from both academia and industry. Meanwhile, bugs in DL systems can be disastrous, and can even threaten human lives in safety-critical applications. To date, a huge body of research efforts have been dedicated to testing DL models. However, interestingly, there is still limited work for testing the underlying DL libraries, which are the foundation for building, optimizing, and running DL models. One potential reason is that test generation for the underlying DL libraries can be rather challenging since their public APIs are mainly exposed in Python, making it even hard to automatically determine the API input parameter types due to dynamic typing. In this paper, we propose FreeFuzz, the first approach to fuzzing DL libraries via mining from open source. More specifically, FreeFuzz obtains code/models from three different sources: 1) code snippets from the library documentation, 2) library developer tests, and 3) DL models in the wild. Then, FreeFuzz automatically runs all the collected code/models with instrumentation to trace the dynamic information for each covered API, including the types and values of each parameter during invocation, and shapes of input/output tensors. Lastly, FreeFuzz will leverage the traced dynamic information to perform fuzz testing for each covered API. The extensive study of FreeFuzz on PyTorch and TensorFlow, two of the most popular DL libraries, shows that FreeFuzz is able to automatically trace valid dynamic information for fuzzing 1158 popular APIs, 9X more than state-of-the-art LEMON with 3.5X lower overhead than LEMON. To date, FreeFuzz has detected 49 bugs for PyTorch and TensorFlow (with 38 already confirmed by developers as previously unknown).

ACM Reference Format:
Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510041

*The work was done during a remote summer internship at University of Illinois.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510041

1 INTRODUCTION

Deep Learning (DL) has been playing a significant role in various application domains, including image classification [39, 62], natural language processing [33, 35], game playing [61], and software engineering [23, 45, 74, 75]. Through such applications, DL has substantially improved our daily life [20, 36, 60, 64, 71]. The great success achieved by DL is attributed to the proposal of more and more advanced DL models, the availability of large-scale datasets, and inevitably, the continuous development of DL libraries. Meanwhile, deploying a DL model without thorough testing can be disastrous in safety-critical applications. For example, a critical bug in the DL system in Uber's self-driving cars has unfortunately taken the life of a pedestrian [12].

Due to the popularity of DL models and the critical importance of their reliability, a growing body of research efforts have been dedicated to testing DL models, with focus on adversarial attacks [15, 22, 34, 50–52] for model robustness, the discussion on various metrics for DL model testing [38, 41, 47, 56, 73], and testing DL models for specific applications [67, 77, 84]. Meanwhile, both running and testing DL models inevitably involve the underlying DL libraries, which serve as central pieces of infrastructures for building, training, optimizing and deploying DL models. For example, the popular PyTorch and TensorFlow DL libraries, with 50K and 159K stars on GitHub, are by far two of the most popular DL libraries for developing and deploying DL models. Surprisingly, despite the importance of DL library reliability, there is only limited work for testing DL libraries to date. For example, CRADLE [57] leverages existing DL models for testing Keras [1] and its backends, and resolves the test oracle problem via differential testing. Later, LEMON [69] further augments CRADLE via leveraging various model mutation rules to generate more diverse DL models to invoke more library code to expose more possible DL library bugs.

Despite their promising results, existing work on testing DL libraries suffers from the following limitations. Firstly, only limited sources for test input generation are considered. For example, CRADLE [57] uses 30 pre-trained DL models and LEMON [69] uses only 12 DL models. Our later empirical results show that they can at most cover 59 APIs for TensorFlow, leaving a disproportionately large number of APIs uncovered by such existing techniques. Secondly, the state-of-the-art model mutation technique proposed by LEMON can be rather limited for generating diverse test inputs for DL APIs.

For example, the intact-layer mutation [69] requires that the output tensor shape of the layer/API to be added/deleted should be identical to its input tensor shape. As a consequence, only a fixed pattern of argument values for a limited set of APIs are explored in model-level mutation, which substantially hinders its bug-finding abilities. Thirdly, model-level testing can be rather inefficient. The inputs for the original/mutated models are obtained from the external DL datasets, and each of them will need to be completely executed end-to-end to get the final prediction results for differential testing, which can consume hours even for a single mutated model. Besides, it requires an additional bug localization procedure to find the specific API invocation causing the inconsistencies between different backends in the original/mutated DL models. During localization, carefully-designed metrics are required to eliminate false positives. The false positives can be due to uncertainty and variances (e.g. floating-point precision loss) in DL APIs, further amplified in the model-level testing scenario.

In this work, we overcome the aforementioned limitations for testing DL libraries via fully automated API-level fuzz testing. Compared with prior model-level DL library testing which resembles *system testing*, API-level testing is more like *unit testing*, which is at a much finer-grained level. The benefit of API-level testing is that it can be a more general and systematic way for testing DL libraries. With API instrumentation, we can get various and diverse input sources from open source to serve the purpose of testing. Moreover, API-level mutation is free of unnecessarily strict constraints on mutation compared with model-level mutation. Besides, API-level mutation neither depends on iterating over external datasets, nor requires complex localization procedures since testing one API at a time does not incur accumulated floating-point precision loss.

One main challenge that we resolve for API-level fuzz testing of DL libraries is fully automated test input generation for DL APIs. The public APIs in DL libraries are mainly exposed in Python, making it difficult to automatically determine the input parameter types due to dynamic typing. To this end, we propose FreeFuzz, the first approach to fuzzing DL libraries via mining from actual model and API executions. More specifically, we consider the following sources: 1) code snippets from the library documentation, 2) library developer tests, and 3) DL models in the wild. FreeFuzz records the dynamic information for all the input parameters for each invoked API on the fly while running all the collected code/models. The dynamic information includes the types, values of the arguments, and the shapes of tensors. The traced information can then form a value space for each API, and an argument value space where values can be shared across arguments of similar APIs during testing. Lastly, FreeFuzz leverages the traced information to perform mutation-based fuzzing based on various strategies (i.e., type mutation, random value mutation, and database value mutation), and detects bugs with differential testing and metamorphic testing on different backends. Our initial evaluation of FreeFuzz on PyTorch and TensorFlow shows that FreeFuzz can automatically trace valid dynamic information for fuzzing 1158 out of all 2530 considered APIs, while state-of-the-art techniques can at most cover 59 APIs for TensorFlow [57, 69]. To date, we have submitted 49 bug reports (detected by FreeFuzz) to developers, with 38 already confirmed by developers as previously unknown bugs and 21 already fixed to date.

In summary, our paper makes the following contributions:

- **Dimension.** This paper opens a new dimension for fully automated API-level fuzzing of DL libraries via mining from actual code and model executions in the wild.
- **Technique.** We implement a practical API-level DL library fuzzing technique, FreeFuzz, which leverages three different input sources, including code snippets from library documentation, library developer tests, and DL models in the wild. FreeFuzz traces the dynamic API invocation information of all input sources via code instrumentation for fuzz testing. FreeFuzz also resolves the test oracle problem with differential testing and metamorphic testing.
- **Study.** Our extensive study on the two most popular DL libraries, PyTorch and TensorFlow, shows that FreeFuzz can successfully trace 1158 out of 2530 APIs, and effectively detect 49 bugs, with 38 already confirmed by developers as previously unknown, and 21 already fixed.

2 BACKGROUND

2.1 Preliminaries for Deep Learning Libraries

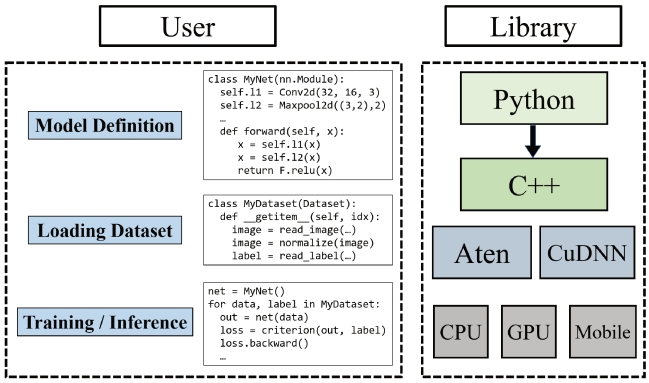

Figure 1: Example DL library (PyTorch)

In this section, we will give a brief introduction to the preliminaries of deep learning libraries based on PyTorch [55].

Training and Inference. As shown on the left-hand side of Figure 1, developers usually leverage DL libraries to support the training and inference tasks on Deep Neural Networks (DNNs). Conceptually, DNNs are composed of *multiple* layers, which are what the adjective "deep" in deep learning refers to. In the model definition part of Figure 1, `Conv2d` and `Maxpool2d` are the APIs invoked to add two layers into the example DNN. Then the `forward` function defines how the input data should flow in the defined layers. Before the actual training and inference, the datasets should also be loaded with necessary pre-processing, e.g., `torchvision.transforms.Normalize` is a crucial step in data pre-processing, which aims to rescale the values of input and target variables for better performance.

Training is the process for a DNN to learn how to perform a task (with its weights updated along the way). For example, for image classification, by feeding a DNN with known data and corresponding labels, we can obtain a trained DL model. Training a DL model involves iterating over the dataset, defining a loss function (e.g., `torch.nn.CrossEntropyLoss`) to calculate the difference between the network outputs and its expected outputs (according to the labels),

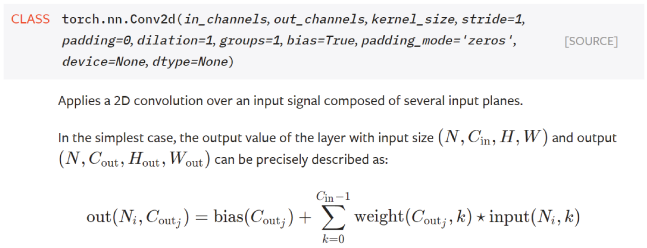

Figure 2: The API definition for 2D-Convolution in PyTorch

and updating the weights of the DNN via a back-propagation procedure (i.e., `loss.backward`). Different from the training phase, inference is the process of using a trained DL model to complete a certain task (with its weights unchanged), e.g., making predictions against previously unseen data based on the trained model.

Abstraction for Hardware. Shown on the right-hand side of Figure 1, DL libraries (such as PyTorch and TensorFlow) usually provide a unified abstraction for different hardware, which can be easily configured by the end users to perform the actual execution. They usually integrate different backends in DL libraries for flexibility and performance. Take PyTorch as an example, Aten [13] is a backend implemented in C++ serving as a tensor library for hundreds of operations. It has specialized low-level implementations for hardware including both CPUs and GPUs. Besides Aten, CuDNN [26] is another backend integrated into PyTorch, which is a widely-used third-party high-performance library, developed specifically for deep learning tasks on Nvidia GPUs. Furthermore, as shown in Figure 1, PyTorch now not only supports general-purpose devices such as CPUs and GPUs, but also allows users to run DL models on mobile devices due to the growing need to execute DL models on edge devices.

2.2 Fuzzing Deep Learning libraries

As shown in the previous subsection, hundreds or even thousands of APIs are implemented in a typical DL library to support various tasks. Therefore, it is almost impossible to manually construct test inputs for each API. Meanwhile, most public APIs from DL libraries are exposed in Python due to its popularity and simplicity, which makes it extremely challenging to automatically generate test inputs given the API definitions. The reason is that we cannot determine the types of the input parameters statically because Python is a dynamically typed language. Take the operator 2D-Convolution from PyTorch as an example, the definition of which is shown in Figure 2, a snapshot captured from Pytorch official documentation [5]. From the definition of 2D-Convolution shown Figure 2, we do not know what types of parameters `in_channels`, `out_channels`, `kernel_size` are. Also, one may conclude that parameter *stride* must be an integer (inferred from the default value `stride=1`) and parameter *padding* must also be an integer (inferred from the default value `padding=0`). However, this is not the case actually. The documentation below (not included in Figure 2 due to space limitations) says that "stride controls the stride for the cross-correlation, *a single number or a tuple*" and "padding controls the amount of padding applied to the input. It can be *either a string 'valid', 'same' or a tuple of ints* giving the amount of implicit padding applied on both sides". In fact, the parameters `kernel_size`, `stride`, `padding`, `dilation` can be either a single *int* or a *tuple of two ints*, and `padding` can even be a *string* besides the two types mentioned above. Therefore, there can exist multiple valid types for a specific argument, and the valid types of arguments cannot be easily inferred from the definition.

Due to the above challenge of test generation for DL APIs, CRADLE [57] proposes to directly leverage existing DL models to test DL libraries. The insight of CRADLE is to check the cross-implementation inconsistencies when executing the same DL models on different backends to detect DL library bugs. It uses 30 models and 11 datasets. After detecting inconsistencies when executing models between different backends by feeding the input from datasets, a confirmation procedure to identify bug-revealing inconsistencies and a localization procedure to precisely localize the source of the inconsistencies have to be launched. In such model-level testing, where inconsistencies can be either due to real bugs or accumulated floating point precision loss propagated through the execution of multiple APIs, carefully designed metrics are needed to distinguish real bugs from false positives. Furthermore, such model-level testing technique only covers a limited range of APIs in DL libraries, e.g., all models used by CRADLE only cover 59 APIs for TensorFlow.

Based on CRADLE, LEMON [69] advances testing DL libraries by proposing model-level mutation. A set of model-level mutation rules are designed to generate mutated models, with the goal of invoking more library code. Model-level mutation is composed of intact-layer mutation and inner-layer mutation. The intact-layer mutation rules pose very strict constraints on the set of APIs to be mutated and the arguments passed to them. As stated in the LEMON paper [69], one explicit constraint for intact-layer mutation is that the output shape of the API to be inserted and the input shape of it must be identical. As a result, only a limited set of APIs with fixed parameters can used for mutation in order to meet such constraints, which substantially hinders LEMON's ability in bug-finding. Moreover, selecting such APIs with specific arguments for layer-level mutation requires expert knowledge of the input-output relation of each API. For example, only a limited range of APIs (e.g., convolution, linear and pooling) with fixed parameters can be added or deleted during model-level mutation. According to our later study, LEMON's various mutation rules can only help cover 5 more APIs in total for all the studied models.

3 APPROACH

Figure 3 shows the overview of our approach, FreeFuzz, which is mainly composed of four different stages. The first stage is code collection (Section 3.1). As shown in the figure, FreeFuzz obtains code from three different sources: 1) code snippets from library documentation, 2) library developer tests, and 3) various DL models in the wild, all of which can be obtained from open source automatically. The second stage is dynamic tracing with instrumentation (Section 3.2). FreeFuzz first hooks the invocation of APIs, and then executes the code collected in the first stage to trace various dynamic execution information for each API invocation, including value and type information for all parameters of all executed APIs. As a result of this stage, FreeFuzz constructs the type space, API value space, and argument value space for the later fuzzing stage. The third stage is mutation-based fuzzing (Section 3.3). Basically,

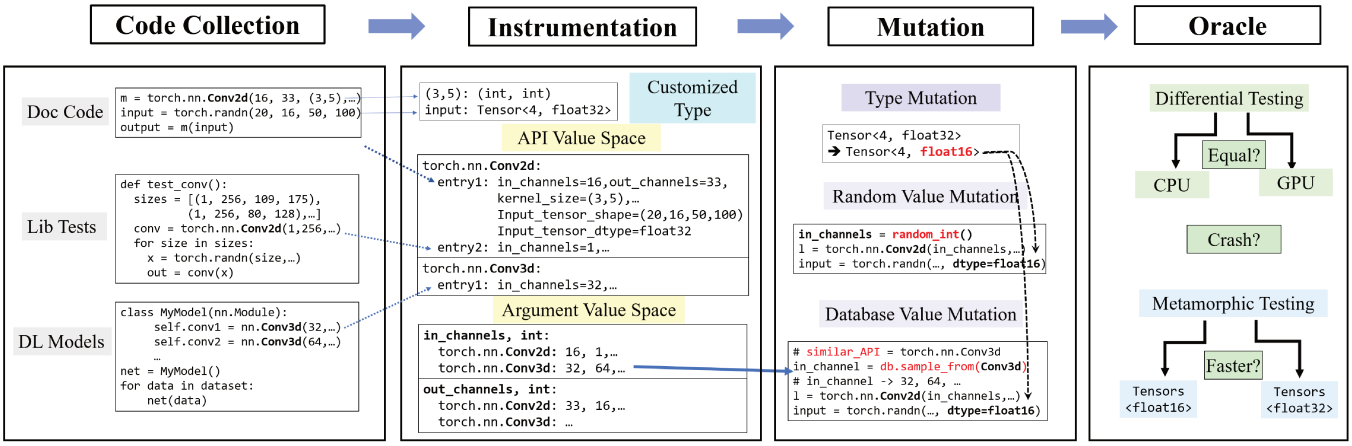

Figure 3: FreeFuzz overview

FreeFuzz effectively generates mutants for the test inputs (i.e., the argument lists) used to invoke the targeted APIs, based on the traced information collected in the second stage. The mutation strategies are composed of type mutation, random value mutation, and database value mutation. The last stage is running all the generated tests with oracles (Section 3.4). FreeFuzz resolves the test oracle problem by differential testing and metamorphic testing on different DL library backends and hardware. FreeFuzz is able to detect various types of bugs, including wrong-computation bugs, crash bugs, and performance bugs for DL libraries.

3.1 Code Collection

FreeFuzz is a general approach and can work with dynamic API information traced from various types of code executions. In this paper, we mainly explore code collection from the following sources:

Code Snippets from Library Documentation. In order to help users better understand the usage of APIs, almost all DL libraries will provide detailed documentations on how to invoke the APIs. Usually, detailed specifications written in natural languages are presented to show the usage of each parameter of each API in detail. Meanwhile, to better help the developers, such natural-language-based specifications are also often accompanied by code snippets for better illustrations. To illustrate, an example code snippet for invoking the 2D-Convolution API within PyTorch is shown in Figure 4. Of course, it is worth noting that not all APIs have example code and example code cannot enumerate all possible parameter values. Therefore, it is also important to consider other sources.

```
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

Figure 4: Example Code for 2D-Convolution from PyTorch's Documentation

Library Developer Tests. Software testing has become the most widely adopted way for quality assurance of software systems in practice. As a result, DL library developers also write/generate a large number of tests to ensure the reliability and correctness of DL libraries. For example, the popular TensorFlow and PyTorch DL libraries have 1493 and 1420 tests for testing the Python APIs, respectively. We simply run all such Python tests as they dominate DL library testing and this work targets Python API fuzzing.

DL Models in the Wild. Popular DL libraries have been widely used for training and deploying DL models in the wild. Therefore, we can easily collect a large number of models for a number of diverse tasks, each of which will cover various APIs during model training and inference. More specifically, from popular repositories in Github, we obtain 102 models for PyTorch, and 100 models for TensorFlow. These models are diverse in that they cover various tasks such as image classification, natural language processing, reinforcement learning, autonomous driving, etc. The detailed information about the leveraged models can be found in our repository [8].

3.2 Instrumentation

In this phase, FreeFuzz performs code instrumentation to collect various dynamic execution for test-input generation. We first get the list of Python APIs to be instrumented from the official documentations of the studied DL libraries in this work, i.e., PyTorch and TensorFlow. More specifically, we hook the invocation of the list of 630 APIs from PyTorch and 1900 APIs from TensorFlow for dynamic tracing. The list includes all the necessary APIs for training and inference of neural networks as well as performing tensor computations. FreeFuzz is able to collect dynamic information for each API invoked by all the three sources of code/model executions, including the type and value for each parameter. No matter how the APIs are invoked (e.g., executed in code snippets, tests, or models), the corresponding runtime information of the arguments is recorded to form the following type/value spaces for fuzzing:

Customized Type Space. FreeFuzz constructs our customized type monitoring system `FuzzType` for API parameters by dynamically recording the type of each parameter during API invocation. Compared with Python's original type system, the customized type system is at a finer-grained level, which can better guide the next mutation phase for fuzzing. In Python's dynamic type system, the type of parameter `stride=(2,1)` (shown in Figure 4) can be calculated by running `type((2,1))`. This will return `<class 'tuple'>`, which does not encode all the necessary useful information for fuzzing because we only know that it is a tuple. In our type monitoring system `FuzzType`, we collect such information at a finer-grained level: `FuzzType ((2,1))` returns `(int, int)` (a tuple of two integers).

Similarly, for tensors, executing `type(torch.randn(20,16,50,100))` simply returns `<class 'torch.Tensor'>` in Python's type system while we can obtain finer-grained type `Tensor<4,float32>` (a 4-dimensional tensor with `torch.float32` as its data type) by running `FuzzType (torch.randn(20,16,50,100))`. Our customized type monitoring system used to guide API-level fuzzing of DL libraries is formally defined in Figure 5.

DT ::=		**Tensor Data Types**
	int32 | int64 | int8 | ...	*int*
	float32 | float16 | tf32 | bfloat16 | ...	*float*
	complex32 | complex64 | ...	*complex*
	...	*other numeric types*
T ::=		**Types**
	tensor<n, DT>	*n-dimensional tensor*
	$(T_i^{i\in 1...n})$	*tuple of typed-values*
	$[T_i^{i\in 1...n}]$	*list of typed-values*
	int | bool | float | str | ...	*Python built-in types*

Figure 5: Customized Type Monitoring System FuzzType

Note that type inference for dynamically typed languages (such as Ruby and JavaScript) via dynamic program tracing has been explored in the literature for traditional applications [16, 17, 58]. In this work, we further extend such techniques for fuzzing deep learning libraries. Different from prior work, FreeFuzz collects dynamic traces from various sources, including developer tests, code snippets documents, and DL models in the wild; also, FreeFuzz augments the Python built-in types to trace and mutate tensor shapes and heterogeneous data types.

API Value Space. FreeFuzz constructs the value space of each API from the concrete values passed into the API during dynamic tracing. One entry in the API value space stands for one API invocation with its corresponding list of concrete arguments, which is later used in our mutation phase as the starting point to generate more mutants/tests. Take Figure 3 as an example, `entry1` is added to the value space of the API `torch.nn.Conv2d` after executing the documentation code in the code collection phase. More specifically, `in_channels=16, out_channels=33, kernel_size=(3,5)` together with some other values (not shown in Figure 3 due to limited space) are recorded in `entry1`. The return value of `nn.Conv2d` is a callable object, and it expects a tensor as its input, which is initialized as `input=torch.randn(20,16,50,100)`, indicating that `input` is a four-dimensional tensor with `(20,16,50,100)` as its shape and the values are randomly initialized. Note that we also record the corresponding shape and data type information for tensors, i.e., `Input_tensor_shape=(20,16,50,100), Input_tensor_type=float32`. All the function arguments mentioned above constitute *one entry* in the value space for `nn.Conv2d`. Each invocation can add a new entry into the value space of the invoked API.

Argument Value Space. As shown in Figure 3, the argument value space is composed of different argument names and types (e.g. `in_channels` of type `int`), together with their values recorded when invoking different APIs. For example, for the argument `in_channels` of the API `torch.nn.Conv2d`, the values recorded include `16,1`, etc. The argument value space is constructed based on the information collected in the API value space to speed up the queries in our database value mutation strategy discussed later. More specifically, argument value space aggregates values from different APIs based on argument names. The argument value space is formed based on the idea that values for an argument of one API can serve as potentially reasonable values for the argument of other similar APIs. For example, `torch.nn.Conv2d` and `torch.nn.Conv3d` can be considered similar. The API definition of 3D-Convolution is `torch.nn.Conv3d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`, and many parameters share the same names as `torch.nn.Conv2d` (shown in Figure 2). The construction of the argument value space is useful for the database value mutation to be introduced in the next section.

3.3 Mutation

In this phase, FreeFuzz applies various mutation rules to mutate the argument values traced in the second phase to generate more tests for fuzzing DL libraries more thoroughly.

Mutation Rules. The mutation rules for FreeFuzz are composed of two parts: *type mutation* and *value mutation*, shown in Tables 1 and 2, respectively. Type mutation strategies include *Tensor Dim Mutation* that mutates n_1-dimensional tensors to n_2-dimensional tensors, *Tensor Dtype Mutation* that mutates the data types of tensors without changing their shapes, *Primitive Mutation* that mutates one primitive type into another, as well as *Tuple Mutation* and *List Mutation* that mutate the types of elements in collections of heterogeneous objects.

Value mutation strategies are divided into two classes: one is random value mutation, and the other is database value mutation. Random value mutation strategies include *Random Tensor Shape* (using random integers as shapes to mutate n-dimensional tensors), *Random Tensor Value* (using random values to initialize tensors), *Random Primitive*, *Random Tuple* and *Random List*. Database mutation strategies include *Database Tensor Shape* and *Database Tensor Value*, which randomly pick the according shapes or values from database of argument value space, together with *Database Primitive*, *Database Tuple*, and *Database List*, which randomly pick the corresponding entries from the argument value space based on the argument names and types. Note that all the mutation rules are type-aware, i.e., they are applied according to the types.

Algorithm. Shown in Algorithm 1, the input to our fuzzing algorithm is the API to be mutated, entries in the API value space `VS`, and the database of argument value space `DB`. Of course, we also need to define the mutation rules as described above. In each iteration, the algorithm always samples the next entry from the API value space `VS[API]` to start the mutation process (Line 3). FreeFuzz then computes the number of arguments `argNum` in the entry (Line 4), and randomly selects an integer between 1 and `argNum` as the number of arguments to be mutated, i.e., `numMutation` (Line 5). Then, FreeFuzz starts an inner loop to mutate `numMutation` arguments to generate a new test. The arguments are mutated one by one via randomly

Table 1: Type Mutation

Mutation Strategies	T_1	T_2
Tensor Dim Mutation	$tensor\langle n_1, DT\rangle$	$tensor\langle n_2, DT\rangle$ $(\lvert n_2 - n_1\rvert > 0)$
Tensor Dtype Mutation	$tensor\langle n, DT_1\rangle$	$tensor\langle n, DT_2\rangle$ $(DT_2 \neq DT_1)$
Primitive Mutation	$T_1 = int \lvert bool \lvert float \lvert str$	T_2 $(T_2 \neq T_1)$
Tuple Mutation	$(T_i^{i\in 1...n})$	$(type\_mutate(T_i)^{i\in 1...n})$
List Mutation	$[T_i^{i\in 1...n}]$	$[type\_mutate(T_i)^{i\in 1...n}]$

Table 2: Value Mutation

Mutation Strategies	T	V
Random Tensor Shape	$tensor\langle n, DT\rangle$	$tensor(shape = [randint()^n], datatype = DT)$
Random Tensor Value	$v : tensor\langle n, DT\rangle$	$tensor(shape = v.shape, datatype = DT).rand()$
Random Primitive	$int \lvert float \lvert bool \lvert str$	$rand(int \lvert float \lvert bool \lvert str)$
Random Tuple	$(T_i^{i\in 1...n})$	$(value\_mutate(T_i)^{i\in 1...n})$
Random List	$[T_i^{i\in 1...n}]$	$[value\_mutate(T_i)^{i\in 1...n}]$
Database Tensor Shape	$tensor\langle n, DT\rangle$	$pick\_shape(database, tensor\langle n, DT\rangle)$
Database Tensor Value	$tensor\langle n, DT\rangle$	$pick\_value(database, tensor\langle n, DT\rangle)$
Database Primitive	$int \lvert float \lvert str$	$pick(database, int \lvert float \lvert str, argname)$
Database Tuple	$(T_i^{i\in 1...n})$	$pick(database, (T_1, T_2, ..., T_n), argname)$
Database List	$[T_i^{i\in 1...n}]$	$pick(database, [T_1, T_2, ..., T_n], argname)$

selecting a random argument index `argIndex` (Line 7). After determining the argument to be mutated each time, FreeFuzz gets the type of it using our customized type system `FuzzType`, the argument name `argName`, and the argument value `argValue` (Lines 8, 9 and 10). The type mutation will be performed nondeterministically – if it is enabled, FreeFuzz will mutate the argument type according to our type mutation strategies (Line 12). `selectRandOverDB` is another random function called to determine whether to perform random value mutation (Line 14) or database value mutation (Line 16) according to the corresponding mutation rules. After mutating `numMutation` arguments for `entry`, FreeFuzz generates a new test, which will be executed for testing the API (Line 19). Then, the main loop will continue to generate the next test until the termination criterion is met, e.g., generating a specific number of new tests.

We next discuss function $ValueRule_{db}$ in more detail to explain the process for mutating the value of an argument for a specific API based on the argument value space. Shown in the algorithm, the function takes the API name `API`, the type of argument `argType`, the name of the argument `argName`, and the database `DB`, as input parameters. It then queries the database to collect all the APIs which share the same argument name and type as the current API under test (Line 21). Next, FreeFuzz computes the text similarities between the current API under test and each of the returned APIs based on the Levenshtein Distance [10] between API definitions (Line 22). Take the query $ValueRule_{db}$`(torch.nn.MaxPool2d, [int, int], 'dilation', DB)` as an example, the text similarity is computed using API definitions of those containing the same argument name (`'dilation'`) and the type (tuple of two integers). More specifically, the similarity between the current API under test and API_i, one of the returned APIs, can be computed by the following formula:

$$Sim(API_i, API) = 1 - \frac{Levenshtein(API_i, API)}{Max(Len(API_i), Len(API))}$$

where function *Levenshtein* computes Levenshtein Distance between the two strings representing API_i and API, and it is divided by the maximum length of the two strings. The whole formula computes the text similarity of the two API definitions. For our example, the result shows that the definition of `torch.nn.Conv2d` has the highest text similarity with the target API `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)`. Then we normalize the text similarities to transform them into probabilities (summing up to 1) for selecting similar APIs (Line 23). The basic idea is that APIs with higher similarity scores should get higher probabilities to be selected. FreeFuzz does this by performing the Softmax computation [14]:

$$Prob(API_i) = \frac{e^{Sim(API_i, API)}}{\sum_{j=1}^{m} e^{Sim(API_j, API)}}$$

where m denotes the number of APIs sharing the same argument name and type as the current API under test. After sampling a random API according to the probabilities (Line 24), the values are then randomly sampled from the values recorded for the API (Line 25). In this way, the values stored in the database from one API can be transferred to serve as the arguments for another API.

3.4 Test Oracle

In this phase, we leverage the following ways to resolve the test oracle problem and detect potential DL library bugs:

Wrong-Computation Bugs. We consider three modes to run each API: CPU, GPU with CuDNN disabled, and GPU with CuDNN enabled. In this way, we can detect wrong-computation results by comparing the results between different execution modes.

Performance Bugs. We leverage metamorphic relations [25, 63] to detect performance bugs with FreeFuzz. More and more data types and hardware accelerators have been proposed in order to boost the DL library performance in recent years. Several floating point data types are specially designed for tensors, including *float32*, *float16*, *tf32*, *bfloat16*, which also appear in our aforementioned tensor data type system. We observe the fact that on the same machine (hardware) $\mathcal{M}$, APIs with the same function arguments *args* and tensors of the same shapes $tensor\langle n, DT\rangle$ tend to hold the following metamorphic relationship in terms of time cost:

$$precision(DT_1) < precision(DT_2) \implies$$
$$cost(\mathcal{M}, API, args, tensor\langle n, DT_1\rangle) < cost(\mathcal{M}, API, args, tensor\langle n, DT_2\rangle)$$

Algorithm 1: Mutation algorithm

```
Input:
  API                                  # the API under test to be mutated
  VS                                   # API value space
  DB                                   # argument value space
Define:
  TypeRule                             # type mutation strategies
  ValueRule_rand                       # random value mutation strategies
  ValueRule_db                         # database value mutation strategies
Function Mutate(API, VS, DB):
    while notFinished do
        entry = selectNext(VS[API])
        argNum = len(entry)            # number of arguments
        numMutation = Random.get(argNum)
        while numMutation > 0 do
            argIndex = selectNext(argNum)
            argType = FuzzType(entry[argIndex])
            argName = entry[argIndex].name
            argValue = entry[argIndex].value
            if doTypeMutation() then
                argType = TypeRule(argType)
            if selectRandOverDB() then
                next = ValueRule_rand(argType, argValue)
            else
                next =
                  ValueRule_db(API, argType, argName, DB)
            entry[argIndex] = next
            numMutation = numMutation − 1
        run(entry)
Function ValueRule_db(API, argType, argName, DB):
    APIs = DB.query(argType, API, argName)
    ⟨API_i, sim⟩ = Sim(APIs, API)
    ⟨API_i, prob⟩ = Softmax(⟨API_i, sim⟩)
    API′ = sample(⟨API_i, prob⟩)
    val = sample(DB, API′, argType, argName)
    return val
```

This indicates that DT_1 carrying less precision information than DT_2 tends to execute faster. For instance, DT_1 can be *float16* while DT_2 is *float32*, as long as the API supports both data types of tensors.

Crash Bugs. If an API crashes or throws runtime exception, then it may be considered as a crash bug. Meanwhile, it could also be due to invalid test inputs which can be generated during the fuzzing process. To automatically filter out such false alarms, we build scripts to heuristically remove crash bugs which throw meaningful exceptions on all backends for invalid inputs, e.g., '`ValueError`', '`InvalidArgumentError`', etc. As a result, if the test program crashes (e.g., segmentation fault), or throws unexpected exception for valid inputs on certain backend(s), it is considered as a crash bug.

4 EXPERIMENTAL SETUP

In the study, we address the following research questions:

- **RQ1:** How do the three different input sources of FreeFuzz (without any mutation) contribute to DL library testing?
- **RQ2:** How does FreeFuzz with different numbers of mutations for each API perform for DL library testing?
- **RQ3:** How do different mutation strategies impact FreeFuzz's performance?
- **RQ4:** How does FreeFuzz compare with existing work?
- **RQ5:** Can FreeFuzz detect real-world bugs?

Our experiments are mainly performed on the stable release versions of DL libraries: PyTorch 1.8 and TensorFlow 2.4. The machine for running experiments is equipped with Intel Xeon CPU (4 cores, 2.20GHz), NVIDIA A100 GPUs, Ubuntu 16.04, and Python 3.9.

4.1 Implementation

Code/Model Collection. Code/model collection is essential to form the original seed test pool for our fuzzing technique. To build an extensive pool, for documentations, we download all 497/512 pieces of code snippets from the official documentations of PyTorch/TensorFlow. More specifically, we use the `bs4` Python package [3] to automatically parse the documentations to obtain the code snippets. Note that not all code snippets crawled from documentations are immediately executable. Thus we also build a simplistic repair tool to insert omitted code in the examples (e.g., `import` sections) to make more code snippets executable. For developer tests, we run all existing Python tests for PyTorch by running `python run_test.py` in the test directory, while for TensorFlow we run all python files with suffix `_test.py`. For DL models, we obtain a diverse set of 102/100 DL models from official model zoos of PyTorch/TensorFlow, and popular GitHub repositories. The detailed information about the models can be found in our repository [8].

Instrumentation. We get the lists of all Python APIs from official documentation of PyTorch and TensorFlow, and hook them in `__init__.py` (a file for a package that will be automatically executed if the package is imported) in the root of the library package by adding a wrapper for each API in the list. This is done transparently and fully automatically for the users so that they do not need to modify any of their code (model code) for instrumentation. In this way, 630 APIs from PyTorch and 1900 APIs from TensorFlow are instrumented for dynamic value tracing. Furthermore, we leverage the MongoDB database [7, 11] to record API value space and argument value space.

Mutation. We implement our main Algorithm 1 for mutation with standard Python packages. The implementation details can also be found in our project repository [8].

Test Oracle. The implementation of differential testing is simple. The example code for PyTorch is shown in Figure 10. Meanwhile, the implementation of metamorphic testing is to wrap the invocation of APIs with code for timing.

4.2 Metrics

To thoroughly evaluate FreeFuzz, we use the following metrics:

Number of Covered APIs. Due to the large number of APIs in DL libraries, it is of great importance to show the number of covered APIs as an important metric of testing adequacy. Surprisingly, such an important metric has been largely overlooked by prior work on DL library testing [57, 69].

Size of Value Space. Each API invocation can add one entry into the API value space. Therefore, we use the total size of value space

for all APIs to serve as the metric to analyze and compare different input sources. To be more accurate, we count the number of entries in the API value space after removing duplicate entries. Please note that this is just used to show the scale of the traced data, and does not necessarily indicate fuzzing effectiveness.

Line Coverage. Code coverage is a widely adopted test adequacy criterion for testing traditional software systems [21, 44, 76] and even the recent tensor compilers [46]. For example, it is impossible for a test to detect bugs in code portions without executing them. Surprisingly, although state-of-the-art DL library testing techniques (e.g., LEMON) claimed to invoke more library code [69], they did not report any code coverage in their experiments. We spent tremendous time and efforts setting up the environment for collecting the most widely used line coverage via GCOV [9] for both PyTorch and TensorFlow. More specifically, we even fixed a bug in the Bazel build system [2] used for building TensorFlow to perform coverage collection. Note that we only trace C/C++ code coverage because the C/C++ implementation provides the fundamental support for operators in DL libraries and almost all the high-level Python APIs finally invoke the C/C++ code.

Number of Detected Bugs. Following prior work on software testing in general and DL library testing [24, 57, 69], we also report the number of actual bugs detected for the studied DL libraries.

5 RESULT ANALYSIS

5.1 RQ1: Input Source Study

In this RQ, we aim to study the effectiveness of directly applying FreeFuzz's traced dynamic information (without any mutation) for testing DL libraries. The main experimental results are shown in Table 3, where we explore different settings, including using documentations only, tests only, models only, and all information together for both TensorFlow and PyTorch. For each setting, we show the number of covered APIs (Row "# API"), the number of traced unique API invocations (Row "# VS"), and the line coverage achieved when directly running the traced API invocations (Row "Line Cov."). From the table, we can observe that different sources of information all tend to be helpful for testing DL libraries. For example, although the test information covers the least number of APIs for TensorFlow, it can still help directly cover 216 APIs and 31293 lines of code; similarly, although the model information covers the least number of APIs for PyTorch, it can still help directly cover 145 APIs and 26292 lines of code. Also, another interesting observation is that the settings covering more APIs tend to also achieve higher code coverage. The reason is that different APIs usually implement different functionalities, and thus usually cover different DL library behaviors/paths. This actually also demonstrates the effectiveness and necessity of API-level testing for DL libraries since it is much easier to cover more APIs at this level than traditional model-level DL library testing [57, 69].

We can also observe that putting all three sources of information together can achieve even better results than using any single source of information. For example, it can cover 470/688 APIs for PyTorch/TensorFlow, and 42425/39575 lines of code for PyTorch/TensorFlow. To better analyze the contribution of each source of information, we further leverage the Venn diagrams in Figure 6 and Figure 7 to present the detailed breakdown of the number of

Table 3: Statistics about different sources

	FreeFuzz PyTorch				FreeFuzz TensorFlow			
	Doc	Test	Model	All	Doc	Test	Model	All
# API	427	176	145	470	486	216	269	688
# VS	1259	3383	10898	15532	1810	6879	36638	45269
Line Cov.	39272	30476	26292	42425	33906	31293	34790	39575

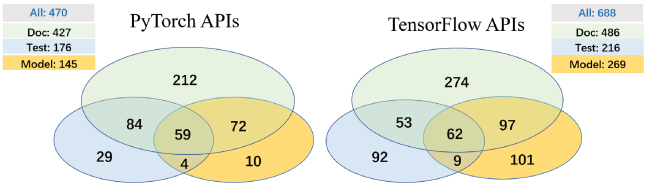

Figure 6: Venn diagram for covered APIs

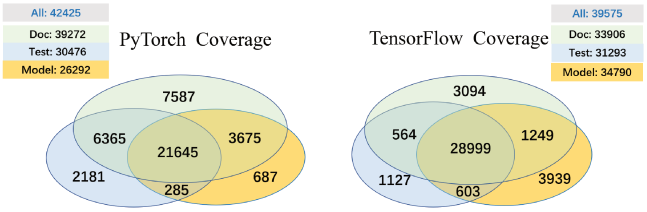

Figure 7: Venn diagram for code coverage

covered APIs and coverage respectively. From the figure, for both TensorFlow and PyTorch, each source of inputs exclusively covers some APIs and only a small number of APIs are covered by all three sources of information. For example, only 59/62 out of all the 470/688 covered APIs are covered by all three sources of inputs on PyTorch/TensorFlow. Meanwhile, although each source of inputs still exclusively covers different code portions, the majority of covered code tends to be shared by all three sources of inputs. The reason is that although different APIs implement different code logic, they can be decomposed to a set of common low-level operators implemented in C/C++. Overall, the experimental results further confirm that it is necessary and important to consider different sources of information for effective DL library testing.

Tracing the three sources of inputs is a *one-time effort* and can be used for testing all subsequent versions of the same DL libraries. Meanwhile, it is also important to demonstrate that the time overhead is acceptable and not extremely high. Therefore, we further discuss the overhead for constructing the three sources of inputs. For the documentation source, the code snippets are usually quite short and fast to run. In total, FreeFuzz takes less than 20 min for tracing all the documentation code snippets for both TensorFlow and PyTorch. For the developer tests, tracing the 1493/1420 official tests written by developers from PyTorch/TensorFlow consumes about 2.5/5.0 hours. Lastly, for the model source, FreeFuzz runs all the 102/100 models stated in Section 4.1 with instrumentation for PyTorch/TensorFlow, consuming less than 1 hour for each of them.

5.2 RQ2: Coverage Trend Analysis

In this RQ, we present the effectiveness of FreeFuzz with different numbers of mutations for each API under test. The experimental results are shown as the blue lines (with legend "*FreeFuzz*") in Figure 8 and Figure 9, where the x axis presents the number of mutants generated for each API (from 100 to 1000 with the interval of 100) while the y axis shows the overall coverage achieved via generating different number of mutants for each API (the union of all coverage

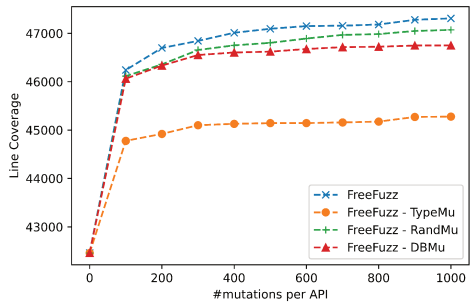

Figure 8: Coverage trend analysis for PyTorch

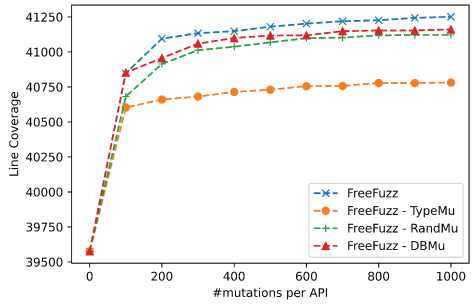

Figure 9: Coverage trend analysis for TensorFlow

sets for all tested APIs). Note that the start point denotes the code coverage achieved by directly executing the original test inputs traced without any mutation. From the figure, we can observe that for both PyTorch and TensorFlow, FreeFuzz can indeed cover more lines of code with more mutations enabled for each API under test, demonstrating the overall effectiveness of our mutation strategies. Furthermore, we can also observe that the coverage becomes largely stable after running 600 mutations for each API, indicating that 600 mutations can be a cost-effective choice in practice. Regarding the time cost, the total running time for generating and running all 1000 mutants for all APIs is 7.3 hours for PyTorch and 9.9 hours for TensorFlow. Note that such overhead is totally acceptable for fuzzing, e.g., traditional binary fuzzing techniques are usually applied for 24h [19] and LEMON takes over 24h [69].

5.3 RQ3: Different Mutation Strategies

After tracing the initial inputs from various sources, FreeFuzz performs three different mutation strategies in tandem (as detailed in Algorithm 1). Therefore, in this RQ, we further study the impact of each mutation strategy by disabling it. To this end, we have three FreeFuzz variants, FreeFuzz-TypeMu (disabling the type mutation strategy), FreeFuzz-RandMu (disabling the random value mutation strategy), FreeFuzz-DBMu (disabling the database value mutation strategy). Note that we also include the variant that does not perform any mutation at all, i.e., FreeFuzz-AllMu. The experimental results for all the studied variants with different number of mutations for each API are also shown in Figure 8 and Figure 9. Note that the start point for all other variants denotes the coverage achieved by FreeFuzz-AllMu. From the figure, we can have the following observations. First, the default FreeFuzz outperforms all the other

Table 4: Comparison on input coverage

	FreeFuzz (tf1.14 full)	LEMON	CRADLE
# API	313	30	59
# VS	9338	1808	2893
Line Cov.	33389	29489	28967

Table 5: Comparison with LEMON on mutation

	FreeFuzz (tf1.14 full)	FreeFuzz (models only)	LEMON
# API	313	30	35
# VS	305463	913	7916
Line Cov.	35473	30488	29766
Time	7h	20 min	25h

studied variants, indicating the importance and necessity of all the three mutation strategies of FreeFuzz. Second, we can also observe that random-value and database-value mutation strategies perform similarly in terms of code coverage, while type mutation can be even more effective since the low-level implementations for different types tend to be more different.

5.4 RQ4: Comparison with Prior Work

In this RQ, we aim to compare FreeFuzz with the state-of-the-art LEMON [69] and CRADLE [57] work for DL library testing. We first compare their sources of inputs in terms of the number of covered APIs and coverage. LEMON only uses 12 models, CRADLE uses 30 models, and FreeFuzz considers three different sources of input with many more models in the wild. Since both LEMON and CRADLE use Keras without supporting PyTorch, the comparison here is only conducted on TensorFlow. Also, due to the fact that LEMON and CRADLE do not support TensorFlow 2 (used in our earlier experiments), we apply FreeFuzz on an old TensorFlow version v1.14. For a fair comparison with prior work, we enforce FreeFuzz to use exactly the same models from LEMON as the DL model input source. To prepare the other two input sources for FreeFuzz, we collect developer tests and documentation code for TensorFlow v1.14. The experimental results are presented in Table 4: Column "FreeFuzz (tf1.14 full)" simply runs the inputs traced by running the same models from LEMON as well as documentation code and tests from TensorFlow v1.14; Columns "LEMON" and "CRADLE" simply run the input DL models used in their original work. From the table, we can observe that, when no mutations are allowed, the input sources used by FreeFuzz can achieve much higher API and code coverage than LEMON and CRADLE.

We next study the effectiveness of the mutation strategies used by FreeFuzz and existing work (i.e., LEMON because CRADLE performs no mutation). We follow the same methodology as the original LEMON work [69] when running its model-level mutations. For FreeFuzz, we also use the default setting, i.e., generating and running 1000 mutants for each covered API. The experimental results are shown in Table 5. Note that besides the default FreeFuzz and LEMON, Column "FreeFuzz (models only)" further includes the results of FreeFuzz with only the models from LEMON (without documentation code and developer tests) as the input for a more thorough comparison with LEMON. From the Table, we can observe that the default FreeFuzz can cover ~9X more APIs than LEMON while consuming ~3.5X less time! Although the coverage improvement is not as significant as the number of covered APIs, FreeFuzz can still outperform LEMON by ~20%. Also, surprisingly, LEMON

```
1  m = torch.nn.Conv2d(64,128,1,2).cuda()
2  tensor = torch.rand(1,64,32,32).cuda()
3  torch.backends.cudnn.enabled = True
4  output1 = m(tensor) # with CuDNN enabled
5  torch.backends.cudnn.enabled = False
6  output2 = m(tensor) # with CuDNN disabled
7  print(output1.sum(), output2.sum()) # debugging
8  assert torch.allclose(output1, output2) # fail
```

Figure 10: Differential testing for 2D-Convolution

only covers 5 more APIs via various model mutations compared to the original models, since only 5 unused layers preserve the strict input-output shape constraints imposed by LEMON and are added into the mutated models. Furthermore, FreeFuzz with models only can already outperform LEMON in terms of code coverage within 20min, i.e., 75X faster than LEMON! This further demonstrates the benefits of API-level testing compared with model-level testing.

5.5 RQ5: Bugs Detected

For bug detection, we target PyTorch 1.8 and TensorFlow 2.4, which are both officially released stable versions, with the default FreeFuzz setting, i.e., generating 1000 mutants for each API. Note that we do not target TensorFlow 1.14 because developers do not actively support it anymore. Table 6 shows the detailed statistics about the real-world bugs detected by FreeFuzz and its various variants studied in Section 5.3. We can observe that FreeFuzz is able to detect 49 bugs in total (with 38 already confirmed as previously unknown bugs) for the two studied DL libraries, and 21 of them have been fixed by the developers to date. Furthermore, we can also observe that each mutation strategy can help detect certain bugs that other mutation strategies fail to detect, further demonstrating the importance of all FreeFuzz mutation strategies. Lastly, of all the 49 bugs detected by FreeFuzz, only one of them can be detected by LEMON and CRADLE.

Table 6: Summary of detected bugs

		FreeFuzz				Confirmed
	FreeFuzz	-TypeMu	-RandMu	-DBMu	-AllMu	(Fixed)
PyTorch	28	13	24	26	5	23 (7)
TensorFlow	21	20	5	20	2	15 (14)

Note that all the detailed issue IDs for the bugs detected can be found on our GitHub page [8]. We next present the case studies:

Wrong-computation Bug. Figure 10 shows an example bug automatically detected by FreeFuzz by comparing the computation results for 2D-Convolution between two backends, one with CuDNN enabled (`output1`) and one disabled, using Aten backend instead (`output2`). It throws `AssertionError` when executing the last line. The sum of values of output tensors in Line 7 shows that `output1 = -523.5300` while `output2 = -601.6165`, indicating a significant difference in computation results. Further comparing the computation results executed by CPU demonstrates that the result is wrong only on GPU with CuDNN disabled. This is a confirmed bug by developers and fixed in latest master.

Performance bug. FreeFuzz detects one performance bug by metamorphic testing for `torch.nn.functional.conv_transpose2d`. According to the metamorphic relations, the time cost for *float16* computation should be less than that for *float32* given the same parameters and tensor shapes. However, our results on NVIDIA A100 GPU

```
1  from torch.nn import Conv3d
2  x = torch.rand(2,3,3,3,3)
3  Conv3d(3,4,3,padding_mode='reflect')(x) # Crash
```

Figure 11: Crash bug in Conv3d

```
1  m_gpu = torch.nn.MaxUnpool2d(2,stride=2).cuda()
2  m_cpu = torch.nn.MaxUnpool2d(2,stride=2)
3  tensor = torch.rand(1, 1, 2, 2)
4  indices = torch.randint(-32768,32768,(1, 1, 2, 2))
5  gpu_result = m_gpu(tensor.cuda(), indices.cuda())
6  cpu_result = cpu(tensor, indices) # only crash on CPU
```

Figure 12: Invalid test input for `torch.nn.MaxUnpool2d`

for PyTorch show that `float16: cost = 0.377s, float32: cost = 0.101s` on some inputs. The bug detected by FreeFuzz has spurred a heated discussion among PyTorch developers. They confirmed this performance bug and are trying hard to figure out the reason.

Crash bug. Figure 11 shows a crash bug detected by FreeFuzz. The program crashes on Line 3 when invoking `torch.nn.Conv3d`. The reason is that argument `padding_mode` is set to value `'reflect'` and the program will not crash if `padding_mode` is set to its default value 'zeros'. The bug is triggered by the database mutation strategy. The argument name `padding_mode` of type string appears in the argument value space, and there exists a value `'reflect'`, which is originally recorded for the argument `padding_mode` of `torch.nn.Conv2d`. FreeFuzz applies the database mutation strategy to query the argument value space, and selects `'reflect'` from `Conv2d` to serve as the input for argument `padding_mode` of `Conv3d`. We confirm this bug according to the documentation of `torch.nn.Conv3d` [6] where 4 string values (i.e., `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`) should be valid for `padding_mode`. Developers have acknowledged this bug and triaged it.

5.6 Threats to validity

The threats to internal validity mainly lie in the correctness of the implementation of our own approach and the compared techniques. To reduce such threats, the authors worked together to perform testing and code review of FreeFuzz; also, we obtained the implementation of prior work from the official websites.

The threats to external validity mainly lie in the evaluation benchmarks used. To demonstrate that our FreeFuzz can be applied/generalized to different DL libraries, we have evaluated FreeFuzz on two most widely used DL libraries, PyTorch and TensorFlow. Furthermore, although FreeFuzz is fuzzing against 1158 APIs (each with 1000 times) and the randomness can be largely mitigated by such a large number of APIs, it is still possible that the nondeterminism in FreeFuzz can affect the effectiveness of FreeFuzz in different runs [18, 42]. Therefore, following existing fuzzing work [59, 70, 83], we rerun the experimental comparison between FreeFuzz and LEMON (Table 5) for 5 times. The results show that FreeFuzz achieves an average line coverage of 35997 (35473 in Table 5), while LEMON's average is 29769 (29766 in Table 5). Both are quite stable with the coefficient of variation of only 0.82%/0.06%, demonstrating the effectiveness of FreeFuzz in different runs.

The threats to construct validity mainly lie in the metrics used. To reduce such threats, we adopt the number of detected bugs used by prior work on DL library testing. Furthermore, we also include the widely used code coverage metric in traditional software testing.

6 DISCUSSION AND FUTURE WORK

Generalizability and Specificity. Different from LEMON [69] and CRADLE [57] that specifically target testing DL libraries via DL models, the FreeFuzz work can potentially be generalized to more than just DL library fuzzing. Of course, in this work, we do have various DL-specific components, including 1) mining DL models as inputs, 2) tensor-related types and mutation rules, and 3) DL-specific oracles (i.e., differential testing for wrong-computation bugs and metamorphic testing for performance bugs). Meanwhile, the basic idea of leveraging code snippets from library documentation and developer tests can be generalized to fuzzing library APIs in various dynamically typed languages. We hope our work can inspire more research on the direction of mining for fuzzing.

Validity of Test Inputs. Our input mining and type-aware [53]/DB-based mutations can all help generate more valid inputs. Meanwhile, FreeFuzz still does not always generate valid inputs due to some complicated input constraints. Interestingly, even the invalid inputs helped detect various bugs in PyTorch/TensorFlow (e.g., unexpected crashes). Figure 12 shows one such bug detected in `torch.nn.MaxUnpool2d`. The input parameter `indices` is a tensor whose values are randomly sampled integers (with respect to the Random Primitive strategy), which is invalid. According to the documentation, the valid `indices` should be obtained from the returned value of `torch.nn.MaxPool2d`. The bug was detected because the program only crashes when running on CPU (i.e., Line 6 fails) but produces a wrong result silently without throwing any error message on GPU (i.e., Line 5 passes). Thus, the GPU implementation should add the missing check. The developers have confirmed this bug and even labelled with "high priority" [4].

Future Work. FreeFuzz currently only focuses on testing the correctness of single APIs. While API-level testing has many advantages over model-level testing, it may still miss bugs which can only be triggered by invoking a sequence of APIs. Besides, when reproducing detected bugs, we find that some tests will fail on one machine but pass on other machines given exactly the same script and the same library version. This is probably due to the differences in underlying infrastructure and hardware. This type of tests are called implementation-dependent flaky tests, described in prior work on test flakiness [43, 54, 78]. Future work may explore how to better detect and fix flaky tests [29–32] in deep learning libraries.

7 RELATED WORK

DL Model Testing. There has been a growing body of research for improving the quality of DL models. Various adversarial attack techniques [34, 51, 52, 79] have been proposed to generate the so-called "adversarial examples" by adding perturbations imperceptible to humans to intentionally fool the classification results given by DL models. To mitigate such attacks, researchers have also proposed various adversarial defense techniques, including adversarial training [34, 50, 66], detection [37, 49, 82], and others [68]. Another recent line of research has explored the possibility of improving the robustness of neural network from a joint perspective of traditional software testing and the new scenario of deep learning. DeepXplore [56] proposes a metric called neuron coverage for whitebox testing of DL models and leveraged gradient-based techniques to search for more effective tests. While various other metrics [41, 47] have also been proposed recently, the correlation between such metrics and the robustness of models is still unclear [27, 38, 73]. Meanwhile, there are also a series of work targeting specific applications, such as autonomous driving, including DeepTest [67], DeepRoad [77], and DeepBillboard [84]. Various techniques have also been proposed to detect numerical bugs introduced when building a DNN model at the architecture level with static analysis [81], and via gradient propagation [72]. Lastly, researchers have also explored concolic testing [65] to achieve higher coverage for DNN models, mutation testing [40, 48] to simulate real faults in DL models, and test input generation for DNNs [28] by exploiting features learned from data with generative machine learning. Different from all such prior work, our work targets the underlying DL libraries, which are the basis for training and deploying various DL models.

DL Library Testing. CRADLE [57] is the trailblazing work for testing DL libraries. The main contribution of CRADLE is resolving the test oracle challenge with differential testing on Keras. LEMON [69] further advanced CRADLE by proposing mutation rules to generate more models, as claimed by LEMON to invoke more code in DL libraries. LEMON's mutation strategies include intact-layer and inner-layer mutation rules, which must conform to strict constraints, e.g., for intact-layer mutation, the layer to be inserted or deleted should preserve the shapes of input tensors. Actually, according to our experimental results, the mutation rules applied by LEMON can hardly help cover more DL library code. A more recent work on testing DL library is Predoo [80], which only mutates the input tensor values with all other API parameters *manually* set up for precision testing. As a result, it was only applied to 7 APIs/operators from TensorFlow and we exclude it in our comparison. To our knowledge, we propose the first general-purpose and fully automated API-level fuzzing approach for popular DL libraries. Furthermore, we adopt traditional code coverage for DL library testing, and reveal various interesting findings (e.g., state-of-the-art LEMON can hardly improve the DL library code coverage).

8 CONCLUSION

We have proposed FreeFuzz, the first approach to fuzzing DL libraries via mining from open source. More specifically, FreeFuzz considers three different sources: 1) library documentation, 2) developer tests, and 3) DL models in the wild. Then, FreeFuzz automatically runs all the collected code/models with instrumentation to trace the dynamic information for each covered API. Lastly, FreeFuzz will leverage the traced dynamic information to perform fuzz testing for each covered API. The extensive study of FreeFuzz on PyTorch and TensorFlow shows that FreeFuzz is able to automatically trace valid dynamic information for fuzzing 1158 popular APIs, 9X more than state-of-the-art LEMON with 3.5X lower overhead. FreeFuzz has detected 49 bugs for PyTorch and TensorFlow (with 38 already confirmed by developers as previously unknown bugs).

ACKNOWLEDGMENTS

We thank Darko Marinov, Chenyang Yang, and Matthew Sotoudeh for their valuable discussions and suggestions. We also appreciate the insightful comments from the anonymous reviewers. This work was partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.

REFERENCES

[1] Keras, 2015. https://keras.io.
[2] Bazel, 2021. https://github.com/bazelbuild/bazel.
[3] bs4, 2021. https://pypi.org/project/bs4.
[4] Bug Report for torch.nn.MaxUnpool2d, 2021. https://github.com/pytorch/pytorch/issues/68727.
[5] Definition of Conv2d from Pytorch official documentation, 2021. https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.
[6] Definition of Conv3d from Pytorch official documentation, 2021. https://pytorch.org/docs/stable/generated/torch.nn.Conv3d.html.
[7] Documentation for PyMongo, 2021. https://pymongo.readthedocs.io/en/stable.
[8] FreeFuzz Repository, 2021. https://github.com/ise-uiuc/FreeFuzz.
[9] GCOV, 2021. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[10] Levenshtein distance, 2021. https://en.wikipedia.org/wiki/Levenshtein_distance.
[11] MongoDB: the application data platform, 2021. https://www.mongodb.com.
[12] News, 2021. https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash.
[13] Pytorch Aten, 2021. https://pytorch.org/cppdocs/#aten.
[14] Softmax function, 2021. https://en.wikipedia.org/wiki/Softmax_function.
[15] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *Ieee Access*, 6:14410–14430, 2018.
[16] J.-h. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. *ACM SIGPLAN Notices*, 46(1):459–472, 2011.
[17] E. Andreasen, C. S. Gordon, S. Chandra, M. Sridharan, F. Tip, and K. Sen. Trace typing: An approach for evaluating retrofitted type systems. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
[18] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE, 2011.
[19] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.
[20] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
[21] B. Brosgol. Do-178c: the next avionics safety standard. *ACM SIGAda Ada Letters*, 31(3):5–6, 2011.
[22] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.
[23] J. Chen, H. Ma, and L. Zhang. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 78–89, 2020.
[24] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan. Practical accuracy estimation for efficient deep neural network testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–35, 2020.
[25] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.
[26] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.
[27] Y. Dong, P. Zhang, J. Wang, S. Liu, J. Sun, J. Hao, X. Wang, L. Wang, J. Dong, and T. Dai. An empirical study on correlation between coverage and robustness for deep neural networks. In *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 73–82. IEEE, 2020.
[28] I. Dunn, H. Pouget, D. Kroening, and T. Melham. Exposing previously undetectable faults in deep neural networks. *arXiv preprint arXiv:2106.00576*, 2021.
[29] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 574–586, 2018.
[30] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 211–224, 2020.
[31] S. Dutta, A. Shi, and S. Misailovic. Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 603–614, 2021.
[32] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 729–739, 2019.
[33] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.
[34] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
[35] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.
[36] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.
[37] S. Gu, P. Yi, T. Zhu, Y. Yao, and W. Wang. Detecting adversarial examples in deep neural networks using normalizing filters. *UMBC Student Collection*, 2019.
[38] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.
[39] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
[40] N. Humbatova, G. Jahangirova, and P. Tonella. Deepcrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 67–78, 2021.
[41] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.
[42] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.
[43] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.
[44] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.
[45] X. Li, W. Li, Y. Zhang, and L. Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.
[46] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *arXiv preprint arXiv:2202.09947*, 2022.
[47] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.
[48] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111. IEEE, 2018.
[49] X. Ma, B. Li, Y. Wang, S. M. Erfani, S. Wijewickrema, G. Schoenebeck, D. Song, M. E. Houle, and J. Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613*, 2018.
[50] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.
[51] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.
[52] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.
[53] J. Park, D. Winterer, C. Zhang, and Z. Su. Generative type-aware mutation for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–19, 2021.
[54] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.
[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.
[56] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.
[57] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038, 2019.
[58] M. Pradel, P. Schuh, and K. Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324. IEEE, 2015.
[59] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray. Mtfuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software*

Engineering, pages 737–749, 2020.

[60] D. Shen, G. Wu, and H.-I. Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[61] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[62] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[63] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–31, 2021.

[64] Y. Sun, D. Liang, X. Wang, and X. Tang. Deepid3: Face recognition with very deep neural networks. *arXiv preprint arXiv:1502.00873*, 2015.

[65] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119, 2018.

[66] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[67] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[68] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1245–1256. IEEE, 2019.

[69] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 788–799, 2020.

[70] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[71] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

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

[73] S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, and X. Zhang. Correlations between deep neural network model coverage criteria and model quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 775–787, 2020.

[74] Y. Yang, X. Xia, D. Lo, and J. Grundy. A survey on deep learning for software engineering. *arXiv preprint arXiv:2011.14597*, 2020.

[75] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang. Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 427–438, 2021.

[76] L. Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209, 2018.

[77] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–142. IEEE, 2018.

[78] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 50–61. IEEE, 2021.

[79] Q. Zhang, Y. Ding, Y. Tian, J. Guo, M. Yuan, and Y. Jiang. Advdoor: Adversarial backdoor attack of deep learning system. 2021.

[80] X. Zhang, N. Sun, C. Fang, J. Liu, J. Liu, D. Chai, J. Wang, and Z. Chen. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 400–412, 2021.

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

[82] Z. Zhao, G. Chen, J. Wang, Y. Yang, F. Song, and J. Sun. Attack as defense: Characterizing adversarial examples using robustness. *arXiv preprint arXiv:2103.07633*, 2021.

[83] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

[84] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 347–358. IEEE, 2020.