# Fuzzing Deep-Learning Libraries via Automated Relational API Inference

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

Chenyuan Yang*
University of Illinois at Urbana-Champaign
cy54@illinois.edu

Anjiang Wei
Stanford University
anjiang@stanford.edu

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

## ABSTRACT

Deep Learning (DL) has gained wide attention in recent years. Meanwhile, bugs in DL systems can lead to serious consequences, and may even threaten human lives. As a result, a growing body of research has been dedicated to DL model testing. However, there is still limited work on testing DL libraries, e.g., PyTorch and TensorFlow, which serve as the foundations for building, training, and running DL models. Prior work on fuzzing DL libraries can only generate tests for APIs which have been invoked by documentation examples, developer tests, or DL models, leaving a large number of APIs untested. In this paper, we propose DeepREL, the first approach to automatically inferring relational APIs for more effective DL library fuzzing. Our basic hypothesis is that for a DL library under test, there may exist a number of APIs sharing similar input parameters and outputs; in this way, we can easily "borrow" test inputs from invoked APIs to test other relational APIs. Furthermore, we formalize the notion of value equivalence and status equivalence for relational APIs to serve as the oracle for effective bug finding. We have implemented DeepREL as a fully automated end-to-end relational API inference and fuzzing technique for DL libraries, which 1) automatically infers potential API relations based on API syntactic/semantic information, 2) synthesizes concrete test programs for invoking relational APIs, 3) validates the inferred relational APIs via representative test inputs, and finally 4) performs fuzzing on the verified relational APIs to find potential inconsistencies. Our evaluation on two of the most popular DL libraries, PyTorch and TensorFlow, demonstrates that DeepREL can cover 157% more APIs than state-of-the-art FreeFuzz. To date, DeepREL has detected 162 bugs in total, with 106 already confirmed by the developers as previously unknown bugs. Surprisingly, DeepREL has detected 13.5% of the high-priority bugs for the entire PyTorch issue-tracking system in a three-month period. Also, besides the 162 code bugs, we have also detected 14 documentation bugs (all confirmed).

*Both authors contributed equally to this research.


*ESEC/FSE 2022, 14 - 18 November, 2022, Singapore*

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

**ACM Reference Format:**
Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *Proceedings of The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Recent years have witnessed the surge of deep learning (DL) in a variety of applications, including computer vision [33, 59], natural language processing [25, 29], robotics [24, 36], bioinformatics [45, 56], and software engineering [26, 30, 37, 38, 44, 67, 76, 77, 79, 80]. Meanwhile, similar as traditional software systems, DL systems can also have bugs, which can lead to serious consequences and may even threaten human lives [4].

To date, most prior work on DL testing focused on testing/verifying DL models, with an emphasis on adversarial attacks [15, 20, 28, 42, 46, 50], metrics for model testing [32, 35, 41, 52, 75], application-specific model testing [61, 83, 86], and verifying certain properties of models [16, 39]. Meanwhile, there is limited work targeting the reliability of DL libraries, which serve as the central infrastructures for building DL models, and are the foundation for running, optimizing and deploying DL models. CRADLE [53] is the trailblazing work for testing DL libraries, which resolves the oracle challenge with differential testing of various DL models on multiple backends of Keras [1]. AUDEE [31] and LEMON [64] further augment CRADLE by leveraging search-based mutation strategies to generate more diverse DL models/inputs for testing library code. Different from the above model-level DL library testing techniques, more recently, FreeFuzz [65] has been proposed to mine example inputs from open source (including code snippets from the library documentation, developer tests, and DL models in the wild) to directly test each DL library API in isolation. FreeFuzz has been evaluated on PyTorch [51] and TensorFlow [14], currently the two most popular DL libraries (with 54K/162K stars on Github). The experimental results show that FreeFuzz can cover 9× more APIs than state-of-the-art LEMON and detect various previously unknown bugs.

Despite the promising results, existing techniques on fuzzing DL libraries still suffer from the following limitations. First, the input generation is still far from optimal. CRADLE and AUDEE can only test APIs that are covered in the original models, and LEMON can cover slightly more APIs with layer mutations; furthermore, although FreeFuzz can cover up to 1158 APIs for PyTorch and TensorFlow (which is already a huge improvement over other work),

it is still unable to test an API if there is no code snippet directly invoking the API. Second, there is still a lack of powerful test oracles. Existing techniques typically perform differential testing across different DL libraries or hardware backends (e.g., GPU/CPU) to address the test oracle issue. However, differential testing across DL libraries is typically applied at the model level and suffers from the limited effectiveness of model-level testing (e.g., limited API coverage and accumulated floating-point precision loss) [31, 53, 64], while different backends often share common code logic/design (and thus may also share similar bugs) [65]. Thus, it is also crucial to investigate novel test oracles for effective DL library fuzzing.

To address the aforementioned limitations, in this work, we open a new dimension for testing DL libraries via automated relational API inference. The inspiration stems from the fact that prior work [17, 27, 43, 63] has discovered a number of equivalent APIs in traditional software systems (e.g., Java projects)[1]. We envision such relational API inference also to be an inspiring direction for fuzzing DL libraries. In this way, given the same inputs generated via fuzzing, APIs that are equivalent in functionality should produce the same numerical result (i.e., *value equivalence*). Moreover, besides the previously studied equivalent APIs, we further leverage the fact that DL APIs with similar functionality should behave similarly in terms of program status (i.e., *status equivalence*) for more effective fuzzing. For example, although `torch.nn.AdaptiveAvgPool3d` and `torch.nn.AdaptiveMaxPool3d` in PyTorch are not equivalent, they are functionally similar APIs; thus, we can feed any valid input of the first API to the second API and expect its invocation to also be successful. Based on this intuition, we can easily "borrow" test inputs generated for one API to test other relational APIs. Also, API relations can directly serve as test oracle for differential testing. Therefore, we can easily overcome the aforementioned limitations.

We have built a fully-automated technique, DeepREL, which infers such API relations without human intervention for fuzzing DL libraries. One key challenge is how to obtain the API relations automatically and accurately. Existing work [17, 27, 43, 63] on equivalent API inference for traditional software systems can hardly be applied for DL library testing, e.g., the most recent MeMo work [17] heavily relies on well-documented API relations, which are rare in DL libraries. To this end, DeepREL first automatically infers all possible candidate matched API pairs based on API syntactic and semantic information. Then, DeepREL synthesizes concrete test programs for those potentially relational APIs. After that, DeepREL leverages a set of representative valid inputs (automatically traced during prior normal API executions) to check whether the inferred API relations hold or not. Lastly, DeepREL takes the validated API pairs, and leverages mutation-based fuzzing to generate a much diverse and extensive set of test inputs for detecting potential inconsistencies among relational APIs. Our study has shown for the first time that there can be a surprising number of equivalent or similar APIs within popular DL libraries (e.g., 4290/8808 verified relational API pairs by DeepREL for PyTorch/TensorFlow), which can substantially help with fuzzing DL libraries (and beyond). In summary, our paper makes the following contributions:

[1]Some of such existing work [17, 27, 43]) treated the entire software systems under test as the test objects, and thus viewed this as *metamorphic testing* [21]. In this paper, we treat each API as a test object and view this as *differential testing* (following [63]).

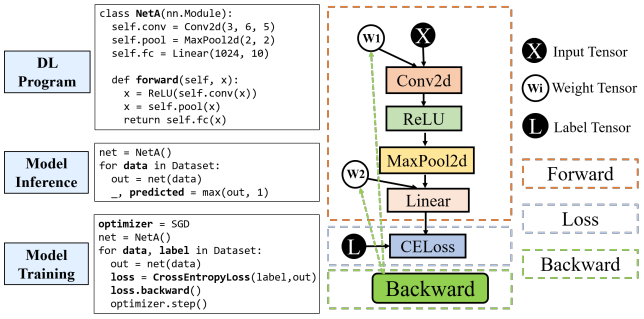



**Figure 1: Background knowledge on DL Models and APIs**

- **Dimension.** This paper opens a new dimension for fully-automated DL library fuzzing via relational API inference.
- **Technique.** We build DeepREL, a *fully-automated* end-to-end framework for DL library testing. DeepREL automatically infers all possible candidate relational APIs based on both API syntactic and semantic information, and then dynamically verifies them via test program synthesis. While this work focuses on DL libraries, the basic idea of DeepREL is general and can also be applied to other software systems.
- **Evaluation and Impact.** DeepREL covers 1815 more APIs than prior work (i.e., 157% improvement), and has detected 162 bugs in total, with 106 already confirmed by the developers as previously unknown bugs. Surprisingly, DeepREL was able to detect 13.5% of the high-priority bugs for the entire PyTorch issue-tracking system in a three-month period. Also, besides the 162 code bugs, we were also able to detect 14 documentation bugs (all confirmed) as a by-product of our experimentation.

## 2 BACKGROUND

### 2.1 Basics about DL Models and APIs

For DL models, *inference* is the process of using a fixed DL model to complete a specific task for unseen data, while *training* is the process for a neural network to update its weights to learn how to better perform a certain task given the labeled data (under the scenario of supervised learning [70]). We next shed light on how this is achieved by APIs from DL libraries and the principles behind.

**DL Models.** To build and run a DL model, developers first need to define the model by writing a DL program in deep learning libraries (e.g., PyTorch [51] and TensorFlow [14]). Take the DL program `NetA` written in PyTorch (shown on the left-hand side of Figure 1) as an example, it includes a convolution layer (`Conv2d`), a max pooling layer (`MaxPool2d`), and a linear layer (`Linear`). The function `forward` defines how the input tensor (`x`) should flow in the defined layers (and other related APIs). Besides input tensors, there are also *weight tensors* (e.g., `W1` and `W2` shown on the right-hand side of Figure 1). Their values will be updated during training, the process of which is called *back-propagation* [69], a procedure natively supported by DL libraries. Training the model is achieved by first running the forward part (i.e., inference) of the neural network (`out = net(data)`), computing the loss (`loss = CrossEntropyLoss(label, out)`), computing the gradient (`loss.backward()`), and invoking the optimizer (`optimizer.step()`) for back-propagation.

**DL APIs.** When running a DL model, the APIs involved in building the model are also executed. Essentially, writing a DL program can be viewed as defining a *computation graph*. It is a directed acyclic graph (DAG) whose nodes stand for DL APIs while edges represent the flow of the tensors. Figure 1 also shows the computation graph for the example neural network. It composes of three parts: forward part (taking input tensors and weight tensors as input), loss computation part (requiring label tensors), and backward part (for updating weight tensors). Actually the backward part needs to construct a much more complex graph [85], but we omit it in Figure 1 for simplicity. Essentially, running a whole DL model can be broken down into invoking a series of DL APIs based on their topological sorting [5] of the computation graph.

## 2.2 Fuzzing DL Libraries

To our knowledge, there are mainly two categories of work for fuzzing DL libraries, model-level testing and API-level testing.

**Model-level Testing.** CRADLE [53] is the first to apply differential testing for DL libraries. Given the fact that Keras [1] is a library featuring high-level APIs for building DL models, APIs in Keras may have multiple implementation in its supported lower-level libraries. Therefore, CRADLE takes 30 pre-trained DL models as input, and runs differential testing to find inconsistencies between different low-level libraries for Keras. More recently, AUDEE [31] and LEMON [64] have proposed to use search-based mutation strategies to generate mutated DL models for differential testing on different backends. While AUDEE focuses on mutating parameters of layers, weight tensors, and input tensors, LEMON applies mutation rules by adding/deleting layers and changing the values of weight tensors. In this way, LEMON's mutation rules are more general, and can cover more APIs than the original DL models. However, even LEMON's model-level mutation can only be applied to a limited number of APIs given a DL model. For instance, the intact-layer mutation rule [64] proposed in LEMON requires that the output tensor shape of the API to be added to (or deleted from) the model should be identical to its input tensor shape. This constraint makes a large number of APIs inapplicable for model-level mutation. Researchers have recently shown that LEMON can hardly invoke additional library code or APIs with its mutation rules [65]. In general, model-level testing suffers from covering only a limited number of APIs, e.g., even state-of-the-art LEMON can only cover 35 APIs for TensorFlow.

**API-level Testing.** Different from prior work on DL library fuzzing, the recent FreeFuzz work [65] proposes to directly mine test inputs from open source for API-level fuzzing, a much finer grained level than model-level testing. One challenge is that Python is a dynamically-typed language, and thus it is hard to determine the types of API parameters for fuzzing Python APIs. Prior work has to manually set up the API arguments, and thus can only test a small number of APIs/operators, e.g., Predoo [84] can only test 7 APIs from TensorFlow. A very recent work DocTer [74] constructs rules to extract DL-specific input constraints from API documentation and uses the constraints to generate valid/invalid inputs for testing DL libraries; meanwhile, it requires manual annotation for 30% of API parameters. In contrast, FreeFuzz resolves this challenge fully automatically via dynamically tracing API executions in code snippets from documentation, developer tests, and 202 DL models. More specifically, FreeFuzz records the traced argument values in a database, and further performs mutation-based fuzzing to mutate those traced values to generate even more inputs for fuzzing DL library APIs. Lastly, FreeFuzz applies differential testing on different hardware backends (i.e., CPU/GPU) for detecting potential consistency bugs. Despite its big improvement over prior work, FreeFuzz can only test 1158 APIs for PyTorch and TensorFlow, which are the ones covered in its input mining stage, leaving a total of 6815 APIs uncovered. Also, different hardware backends may still share code logics/design, causing the differential testing oracle used by FreeFuzz to miss various bugs. In this work, we propose to test relational APIs to further overcome such limitations. We build our technique (DeepREL) upon FreeFuzz to automatically infer relational APIs and leverage them to fuzz DL libraries. Note, however, that our DeepREL idea is general and can be built on any other API-level fuzzer for DL libraries (e.g., DocTer [74]). We choose FreeFuzz since it is a recent state-of-the-art technique that is both publicly available and fully automated.

```
result1 = torch.broadcast_shapes(*shapes)
result2 = torch.broadcast_tensors(*map(torch.empty, shapes))[0].shape
```

**Figure 2: API pair with the same output**

```
layer1 = torch.nn.AdaptiveAvgPool3d(output_size)
result1 = layer1(input)
layer2 = torch.nn.AdaptiveMaxPool3d(output_size)
result2 = layer2(input)
```

**Figure 3: API pair with different outputs but same status**

# 3 PRELIMINARIES

We first introduce the preliminaries for our fuzzing technique in this section. Given the set of all possible APIs, $\mathcal{A}$, for a DL library under test, we aim to define the relational property between the invocation results of a source API $S \in \mathcal{A}$ and a target API $T \in \mathcal{A}$.

Intuitively, we can directly check whether $S$ and $T$ produce equivalent outputs. For example, Figure 2 shows an API pair which, according to the PyTorch documentation [3], should always produce the same results. The `torch.broadcast_shapes` API applies broadcasting on a list of compatible shapes to align them. The `torch.broadcast_tensors` API applies broadcasting on a list of shape-compatible tensors to align their shapes. In fact, the first API can be rewritten as 1) creating intermediate empty tensors from tensor shapes with `map` and `torch.empty`, 2) calling `torch.broadcast_tensors` with these tensors, and 3) getting the shape of the output tensor. Since the source and target APIs can achieve the same functionality with totally different implementations given the same input (`shapes` in Figure 2), they provide a great opportunity for differential testing. Therefore, we have the following formal definition:

*Definition 3.1.* **Equivalence$_{value}$.** Given a set of inputs $\mathcal{D}$, source API $S \in \mathcal{A}$ and target API $T \in \mathcal{A}$ satisfy Equivalence$_{value}$ (modulo $\mathcal{D}$) iff their invocations always output the same results given any input in $\mathcal{D}$. Formally,

$$S \equiv T (mod\ \mathcal{D}) \iff \forall x \in \mathcal{D}.\ S(x) = T(x) \tag{1}$$

While this can be effective in detecting potential consistency bugs, the checking is too strict and may not apply to a large number of APIs. In fact, it could be possible that $S$ and $T$ produce totally different results, but tend to behave similarly given similar inputs. For example, the API pair shown in Figure 3 does not hold the Equivalence$_{value}$ property since the output of `AdaptiveAvgPool3d` is different than `AdaptiveMaxPool3d`. The first API applies a 3D adaptive *average* pooling over an input but the latter applies a 3D adaptive *maximum* pooling. However, these two APIs do have something in common in terms of functionality that both of them apply a pooling operation, which is also valuable for testing. Therefore, we further abstract the invocation results of a program into a set of coarse-grained *statuses*: `Success`, `Exception`, and `Crash`. `Success` denotes that program executions terminate normally, while `Exception` means that program executions throw known exceptions. Lastly, `Crash` represents the cases where the program executions crash with unexpected errors, e.g., segmentation faults or `INTERNAL ASSERT FAILED` errors (which are "*never acceptable*" as commented by PyTorch developers). We then further introduce the notation of $[\![\cdot]\!] \in \{\texttt{Success}, \texttt{Exception}, \texttt{Crash}\}$ to return the execution status of the input program. For example, $[\![S(x)]\!] = \texttt{Success}$ indicates that $S$ terminates normally with input $x$. In this way, we can define another property for checking potential consistency:

*Definition 3.2.* **Equivalence$_{status}$.** Given a set of inputs $\mathcal{D}$, source API $S \in \mathcal{A}$ and target API $T \in \mathcal{A}$ satisfy Equivalence$_{status}$ (modulo $\mathcal{D}$) iff their invocation always output the same statuses given any input in $\mathcal{D}$. Formally,

$$S \sim T (mod\ \mathcal{D}) \iff \forall x \in \mathcal{D}.\ [\![S(x)]\!] = [\![T(x)]\!] \tag{2}$$

To conclude, the Equivalence$_{status}$ relation is a relaxed notation for the Equivalence$_{value}$ relation, which is in turn a relaxed notation of semantic equivalence (denoted as $S \equiv T$). Formally,

$$S \equiv T \implies S \equiv T (mod\ \mathcal{D}) \implies S \sim T (mod\ \mathcal{D}) \tag{3}$$

One crucial component of these two definitions is the domain $\mathcal{D}$ on which the properties are constrained on. Aiming for more accurate API relations, it would be beneficial to cover more representative test inputs within the intersection of the valid input space of the source and target APIs as the domain.

# 4 FUZZING RELATIONAL APIS

Figure 4 shows the overview of our DeepREL technique for fuzzing relational APIs of DL libraries. DeepREL takes as input the targeted DL library, its API documentation, and a database of valid historical API invocations (e.g., automatically collected via running documentation examples, library tests, and DL models [65]). Each entry of the database contains the concrete argument values passed into an API during invocation, and is obtained through dynamic tracing. Overall, DeepREL performs the following four phases iteratively:

*API Matcher (Section 4.1).* In order to test a DL library which typically has hundreds or even thousands of APIs, the first challenge is to identify the API pairs that are likely to satisfy the desired properties Equivalence$_{value}$ or Equivalence$_{status}$. API Matcher maps each API into embeddings based on API documentation, and uses embedding similarity to identify candidate matched APIs.

*Invocation Synthesizer (Section 4.2).* Given a collection of potential matched API pairs, Invocation Synthesizer decides how to invoke them. To construct valid invocations for later verification, we impose a constraint on the source API: it must have at least one (valid) invocation in the database. In this way, given an invocation of the source API, Invocation Synthesizer aims to synthesize the invocation code for the target API.

*API Match Verifier (Section 4.3).* Given the invocation code of matched APIs, this phase would check whether each API pair satisfies property Equivalence$_{value}$ or Equivalence$_{status}$ with a set of representative inputs as the verifying test inputs. If the result values (resp. execution statuses) are consistent for all tests, then API Match Verifier accepts the API pair as Equivalence$_{value}$ (resp. Equivalence$_{status}$). If API Match Verifier detects any inconsistency in this phase, it then rejects the API pair.

*API Fuzzer (Section 4.4).* The last step is to leverage the verified API pairs to detect potential consistency bugs. API Fuzzer uses mutation-based fuzzing to generate a large number of test inputs for source APIs, and tests the verified API pairs with oracles (Section 3).

Lastly, recall that in order to generate valid inputs, the source API must have at least one (valid) invocation in the database. DeepREL further adopts an iterative process to cover more API pairs (Section 4.5). The newly generated valid target API invocations can be added to the database to serve as the source APIs for the next iterations to detect more potential API pairs. The following sections would explain each phase in detail.

## 4.1 API Matcher

In this phase, DeepREL identifies potential matched API pairs from documentation. DeepREL uses API Matcher to infer similar API pairs as matched API pair candidates (which will be further verified later). API Matcher would map each API into API embeddings, and compute similarities of each API pair to be the distance of their embeddings. We consider *signature similarity* and *document similarity* that cover both API syntactic and semantic information for similarity computation. Overall, for each API pair, the similarity is defined as the maximum of the two:

$$Sim_{API}(S,T) = Max(Sim_{sig}(S,T), Sim_{doc}(S,T)) \tag{4}$$

We compute the pair-wise similarity for every API pair, and pair each API with its $K$-closest neighbours as the candidate matched API pairs. $K$ is a hyper-parameter and it is set to 10 in the default setting of DeepREL. Notably, we also analyze the impact of different values of $K$ in our experimental study (Section 6.3).

**Signature Similarity.** The signature of an API contains the API name and an ordered list of argument names. The APIs to be paired tend to follow a similar syntactic pattern in terms of their signature. For example, `tf.math.maximum` and `tf.math.minimum` are two APIs satisfying Equivalence$_{status}$, and their signatures are very similar: `tf.math.maximum(x, y, name=None)` and `tf.math.minimum(x, y, name=None)`. We map an API signature into its TF-IDF (term frequence - inverse document frequency) embedding [72], and use the embedding distance as the similarity measure.

TF-IDF has been widely adopted in the field of information retrieval, and it reflects the importance of each word in a document. Some common words like `tf` and `torch` in the API signature are less informative, so their TF-IDF weights tend to be smaller. To obtain the TF-IDF embedding for each API, we first break the API signatures into subwords (also called tokens) and then standardize

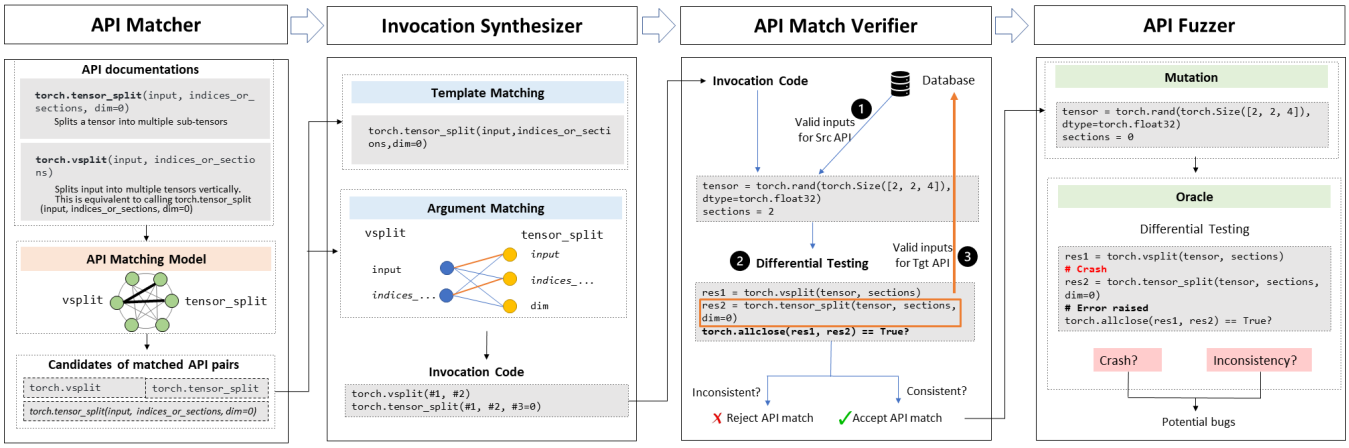



**Figure 4: DeepREL overview**

them. Let $n$ denote the size of the vocabulary from all the tokenized API signatures, an API $S$ can be represented as an unnormalized term frequency embedding $[c_1^S, c_2^S, \ldots, c_n^S]$, where $c_j^S$ is the number of occurrences of word $j$ in the API signature of $S$. We further normalize it with the inverse document frequency for each word to get the TF-IDF embedding:

$$Rep_{sig}(S) = [\frac{c_1^S}{\sum_{S' \in \mathcal{A}} c_1^{S'}}, \frac{c_2^S}{\sum_{S' \in \mathcal{A}} c_2^{S'}}, \ldots, \frac{c_n^S}{\sum_{S' \in \mathcal{A}} c_n^{S'}}] \quad (5)$$

The cosine similarity of two vectors is the cosine of the angle between them, and thus always belongs to the interval $[-1, 1]$. The cosine similarity of two arbitrary vectors $x, y$ is defined as follows:

$$Cos(x, y) = \frac{x \cdot y}{\|x\|\|y\|} \quad (6)$$

We then compute the cosine similarity between the TF-IDF embeddings to be the signature similarity of two APIs $(S, T)$:

$$Sim_{sig}(S, T) = Cos(Rep_{sig}(S), Rep_{sig}(T)) \quad (7)$$

**Document Similarity.** To complement the signature similarity, we further model the semantic similarity between API documents. We extract API descriptions from API documents, each of which is a one-sentence summary of an API given in the beginning of the document. For example, API `torch.vsplit` is described as "*Splits input, a tensor with two or more dimensions, into multiple tensors vertically according to indices_or_sections.* " The description succinctly and surgically states the expected input, the transformation applied, and the expected output. We use Sentence-BERT [57] to encode these informative description sentences into semantically meaningful sentence embeddings. Sentence-BERT targets specifically on generating embeddings whose cosine distance reflects the semantic textual similarity. We first collect all API descriptions from the documentation. The description for an API $S$ is denoted as $S.description$. We use $SBEncoder$ to denote the Sentence-BERT encoder, which takes a natural language sentence as input, and outputs a vector in high-dimensional space. For each API $S$, we encode it with $SBEncoder$ to obtain its document embedding:

$$Rep_{doc}(S) = SBEncoder(S.description) \quad (8)$$

For each API pair $(S, T)$, we compute the cosine similarity between their document embeddings as their document similarity:

$$Sim_{doc}(S, T) = Cos(Rep_{doc}(S), Rep_{doc}(T)) \quad (9)$$

## 4.2 Invocation Synthesizer

In this phase, we leverage argument matching and template matching to synthesize the invocation code for each matched API pair. Note that the invocation code of source API is simply the code snippet that directly invokes the source API with the valid traced inputs. Therefore, we will next focus on generating the target API invocation code.

**Argument Matching.** For each matched API pair candidate, DeepREL first synthesizes the invocation code based on API definitions. It maps the arguments of the source API to the arguments of the target API to synthesize the invocation code of the target API (with the arguments from the source API).

We transform the argument matching problem into a *maximum weighted bipartite matching* problem [68]. DeepREL generates the invocation code based on the best argument match. More formally, given the source API $S$, the target API $T$, and their argument lists $S.args$ and $T.args$, the corresponding bipartite graph is $G = (L, R, E)$, where $L = \{a \mid a \in S.args\}$, $R = \{b \mid b \in T.args\}$ and $E = \{(a, b) \mid a \in L, b \in R\}$. The weight of each edge $(a, b) \in E$ is the similarity $Sim_{arg}(a, b)$ of $a$ and $b$, which is defined as:

$$Sim_{arg}(a, b) = Sim_{name}(a, b) + Sim_{type}(a, b) + Sim_{pos}(a, b) \quad (10)$$

The similarity $Sim_{arg}(a, b)$ is determined by the names, potential types, and positions of the arguments. First, the similarity of two argument names is computed based on the following formula:

$$Sim_{name}(a, b) = 1 - \frac{Levenshtein(a_{name}, b_{name})}{Max(Len(a_{name}), Len(b_{name}))} \quad (11)$$

where $a_{name}$ is the name of argument $a$. This is based on the Levenshtein Distance [71] between two names. Next we compute the similarity of two type sets as:

$$Sim_{type}(a, b) = \frac{|a_{type} \cap b_{type}|}{|a_{type}|} \quad (12)$$

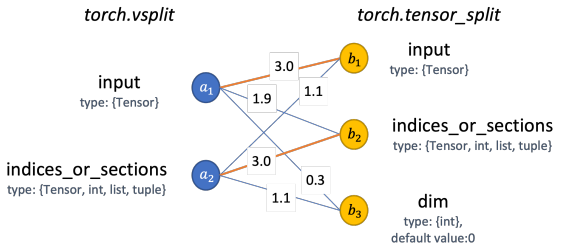



**Figure 5: Example weighted bipartite graph**

```
torch.vsplit(#1, #2)
torch.tensor_split(#1, #2, dim=0)
```

**Figure 6: Invocation synthesis via argument matching**

where $a_{type}$ is the set of possible types of $a$. If the set of types that argument $b$ can take contains all possible types of $a$, $a$ is more likely to be mapped to $b$ since all types of $a$ are legal for $b$.

We also compute the positional similarity of two arguments as:

$$Sim_{pos}(a,b) = 1 - \frac{|a_{idx} - b_{idx}|}{Max(Len(S.args), Len(T.args))} \quad (13)$$

where $a_{idx}$ is the index of $a$ in $S$'s argument list. For example, if $a$ and $b$ are both the first argument for $S$ and $T$, then $|a_{idx} - b_{idx}|$ equals to 0 and thus their positional similarity is 1.

After constructing this graph, DeepREL leverages the Kuhn–Munkres algorithm [47] to find the best argument match and synthesizes the invocation code based on it. When the source and target APIs have the same number of arguments, DeepREL will synthesize the invocation code based on the best argument match directly. Otherwise, when they have different numbers of arguments, if the unmatched arguments contain non-optional arguments, argument matching will abort for the current API pair since the search space for determining the values of those unmatched non-optional arguments is huge. That said, DeepREL only considers the case where the optional arguments of the source or target API are not matched. For the unmatched optional arguments, DeepREL just uses their default values (Python optional arguments always have default values).

For instance, consider API pair `torch.vsplit` and `torch.tensor_split`. The corresponding weighted bipartite graph is shown in Figure 5. For each vertex, its name and type information are marked next to it, such as vertex $a_1$, whose argument name is `input`, possible type set is composed of `Tensor`, and index is 1. The weight of each edge, that is, the similarity of the two arguments, is marked on the corresponding edge. For vertices $a_1$ and $b_1$, because they have exactly the same name, possible type and index, the similarity between them is 3. For vertices $a_1$ and $b_2$, their names are different but the type that $a_1$ can take is legal for $b_2$, so they have a relatively high similarity of 1.9. However, for $a_2$ and $b_1$, only one type of $a_2$ is legal for $b_1$, which causes them to have a low similarity of 1.1. Overall, the best match in the graph is $\{(a_1, b_1), (a_2, b_2)\}$, leaving $b_3$, the optional argument `dim` of `torch.tensor_split`, unmatched. Then DeepREL will set `dim` as its default value 0 to generate the invocation code, as shown in Figure 6 (where placeholders `#i` indicate the argument mapping between source and target APIs).

Note that for every argument of every API, we gather the possible types that it can take in the invocation traces from open source. To be precise, we extract the argument type information from all the invocation cases of the API to form the possible type set. The type set depends on the traces and therefore is likely to be incomplete. When no traces cover a particular argument $b$ of a target API, the type set $b_{type}$ is an empty set; thus, the type similarity between $b$ and any other argument $a$ will be $Sim_{type}(a,b) = 0$. Note that the argument matching algorithm still works in this scenario, with only the name and positional similarities being considered.

```
tf.scatter_nd(indices, updates, shape, name=None)
    Calling tf.scatter_nd(indices, updates, shape) is identical to calling
tf.tensor_scatter_nd_add(tf.zeros(shape, updates.dtype), indices, updates).
```

**(a) Documentation of `tf.scatter_nd`**

```
tf.scatter_nd(#1, #2, #3)
tf.tensor_scatter_nd_add(tf.zeros(#3, #2.dtype), #1, #2)
```

**(b) Invocation code from template**

**Figure 7: Invocation synthesis via template matching**

**Template Matching.** For some complex matched API pairs, DeepREL cannot leverage argument matching to generate the correct invocation code of target API. For example, Figure 7b shows the right invocation code between `tf.scatter_nd` and `tf.tensor_scatter_nd_add`. It is obvious that argument matching fails to synthesize the invocation code of `tf.tensor_scatter_nd_add`. In this case, a matching template can help DeepREL synthesize correct invocation code. A *matching template* is a code snippet elaborating a matched API pair. It presents an invocation of the target API, whose inputs are obtained from the arguments of the invocation of source API. Figure 7a shows an example matching template (highlighted with underline) from the documentation of `tf.scatter_nd`. It suggests that invoking `tf.tensor_scatter_nd_add` is equivalent to invoking `tf.scatter_nd` with proper argument mappings as shown in Figure 7b.

To automatically detect such templates, DeepREL examines each code block in the documentation. Whenever a code snippet contains the invocation of another API, DeepREL extracts it as a potential candidate. Note that not every API pair has such templates. If DeepREL failed to extract any, the matching template will simply be `None`. For API pairs with a matching template, regardless of whether it is in top-$K$ similar pairs, DeepREL synthesizes additional invocation code using the matching template as the target API invocation code.

## 4.3 API Match Verifier

In this phase, DeepREL runs the invocation code synthesized for each matched API pair over a set of *verifying inputs* to validate properties $\text{Equivalence}_{value}$ and $\text{Equivalence}_{status}$ (defined in Section 3). The verifying inputs are a collection of valid inputs for the source API. These inputs can be collected from documentations, library tests, and existing DL models, so they are representative of the valid input space of the source API. For example, in our implementation, we leverage state-of-the-art FreeFuzz [65], which can collect all such information fully automatically.

**$\text{Equivalence}_{value}$.** DeepREL first checks whether the invocation code holds the $\text{Equivalence}_{value}$ property given the verifying inputs. If so, DeepREL accepts this matching pattern and marks it as $\text{Equivalence}_{value}$. For example, the invocation code in Figure 7b holds this property. For all verifying inputs, `tf.scatter_nd` always

have the same output with `tf.tensor_scatter_nd_add`. Hence, this API pair is labeled as Equivalence$_{value}$.

**Equivalence**$_{status}$. If the invocation code violates Equivalence$_{value}$, DeepREL checks whether the source API and target API in the invocation code have the same status given verifying inputs. If so, DeepREL accepts this API pair and marks it as Equivalence$_{status}$. For instance, the invocation code shown in Figure 3 does not hold the Equivalence$_{value}$ property since the output of `AdaptiveAvgPool3d` is different from `AdaptiveMaxPool3d` over the input set. However, they have the same status over the verifying inputs, which allows them be labeled as Equivalence$_{status}$.

If the API pair is verified as Equivalence$_{value}$ or Equivalence$_{status}$, the API pair that is accepted by the API Match Verifier will be further tested in the next fuzzing phase; otherwise, DeepREL rejects this pair. Please note that it is crucial to have a set of representative verifying inputs. If the verifying inputs are not representative, the API Match Verifier could mistakenly accept a wrong API relation when the verifying inputs do not cover certain important regions of the possible input space. We will study such false positives in detail in our experimental study (Section 6.2). On the other hand, the API Match Verifier may also reject some true matched API pairs if the verifying inputs directly trigger real consistency bugs in this phase. Although it is hard to avoid such false negatives, our experimental results show that DeepREL detects 162 bugs for popular DL libraries fully automatically, demonstrating the effectiveness of this design.

### 4.4 API Fuzzer

In the last phase, DeepREL further leverages the verified API pairs to detect bugs for DL libraries. Specifically, DeepREL applies off-the-shelf mutation-based fuzzing techniques [65] to mutate the source API inputs for generating diverse inputs for differential testing. For Equivalence$_{value}$, the source and target APIs are expected to have the same output. We can detect potential consistency bugs by comparing the detailed results of the source and target APIs. For Equivalence$_{status}$, the source and target APIs are expected to have the same status. Therefore, we can detect bugs by comparing their statuses after execution.

### 4.5 The Iterative Process

In order to cover more APIs by matched API pairs, DeepREL performs the above four phases iteratively until the fixed point or a given number of iterations $I$ (by default $I = 10$ for this paper). In the API Match Verifier phase, if the target API is not covered in the current iteration and its invocation generated by synthesizer has `Success` status, we will add this invocation into the database and label the target API as "newly covered API". After this iteration, if there is any newly covered API, DeepREL will re-run the framework *with these newly covered APIs as the source APIs*; otherwise the fixed point has been reached, and DeepREL will terminate. It is worth mentioning that the entire iterative DeepREL approach is fully automated. For the newly covered APIs, the verifying inputs are also automatically borrowed from the source APIs' valid inputs.

Figure 4 presents one example for this iterative process. In the first iteration, the API Match Verifier takes API pair (`torch.vsplit`, `torch.tensor_split`) as input, and queries the invocation database ❶ to get a record for `torch.vsplit`. Invoking `torch.tensor_split` ❷ results in `Success`. Assuming that `torch.tensor_split` is not covered in the database at the beginning of the iteration, this successful invocation is then inserted into the database ❸. In the next iteration, this invocation record will be retrieved ❶ to verify the matched API pairs with `torch.tensor_split` as the source API.

## 5 EXPERIMENTAL SETUP

In the experiments, we address the following research questions:

- RQ1: How effective is DeepREL in terms of API coverage?
- RQ2: What is the false positive rate of DeepREL?
- RQ3: How do different configurations affect DeepREL?
- RQ4: Can DeepREL detect real-world bugs?

Our experiments are performed on PyTorch 1.10 [51] and TensorFlow 2.7 [14], the latest stable release versions for the two most popular DL libraries, with 54K and 162K stars on GitHub. They have also been the most widely studied DL libraries in prior DL library testing work [31, 53, 64, 65]. The machine for running the experiments is equipped with 8-core 2.20GHz Intel Xeon CPU, 16GB RAM, Ubuntu 20.04, and Python 3.9.

### 5.1 Implementation

**API Matcher**. This phase provides candidates of matched API pairs for later verification and fuzzing. To find high-quality matched API pairs, we first use the bs4 Python package [2] to parse the documentations of all 7973 APIs from TensorFlow and PyTorch. We collect both API signatures and descriptions from the documentation. To compute the TF-IDF embedding, we use the Snowball stemmer [54] to convert tokens into word stems. To compute the document embedding, we use the SentenceTransformer Python package [13] and use the pretrained model `all-MiniLM-L6-v2` as our *SBEncoder*.

**Invocation Synthesizer**. For argument matching, we use the munkres Python package [10] (which implements the Kuhn–Munkres algorithm) to solve the maximum weighted bipartite matching problem. For template matching, we automatically search and extract the code snippets for matching template from the documentation.

**API Match Verifier**. We verify each invocation code with the help of verifying inputs. We obtain the valid inputs traced from various input sources used by state-of-the-art FreeFuzz [65], which include documentations, developer tests, and 202 DL models in the wild, and are representative to verify the function of APIs. We feed the first 100 valid invocations of the source API from the FreeFuzz database into both the source API and target API as the verifying inputs, and check if they have consistent behaviors. If there are fewer than 100 records for the source API, we use all of them.

**API Fuzzer**. We leverage the fuzzing strategies of FreeFuzz to mutate all the valid inputs traced for each source API (within the FreeFuzz database), and run all the generated inputs (1000 for each source API following the default setting of FreeFuzz [65]) on both the source API and target API for detecting consistency bugs.

### 5.2 Metrics

**# of Covered API**. Following prior work in DL library testing [65], we report the number of covered APIs. An API is covered by DeepREL if it is successfully invoked by API Match Verifier either as a source API or a target API (i.e., invocations with the `Success`

**Table 1: Comparison with FreeFuzz on API coverage**

| | #Total | #FreeFuzz | #DeepREL | Improvement(%) |
|---|---|---|---|---|
| PyTorch | 1592 | 470 | 1071 | 601 (128%) |
| TensorFlow | 6381 | 688 | 1902 | 1214 (176%) |
| Total | 7973 | 1158 | 2973 | 1815 (157%) |

**Table 2: Verified API pairs**

| | Equivalence$_{value}$ | Equivalence$_{status}$ | Total |
|---|---|---|---|
| PyTorch | 1357 | 2933 | 4290 |
| TensorFlow | 5132 | 3676 | 8808 |
| Total | 6489 | 6609 | 13098 |

status). Since DL libraries contain a large number of APIs, API coverage is an important metric of test adequacy.

**False Positive Rate**. If an API pair satisfies that 1) at least one of its invocation code is accepted by the API Match Verifier, and 2) at least one of its accepted invocation code is against its labeled property during the fuzzing phase, it is named an inconsistent API pair. False positive rate for inconsistent API pairs is the proportion of detected inconsistent API pairs which are false alarms. It is commonly used in prior work on fuzzing or testing [23, 60, 78].

**# of Detected Bugs.** Bug finding is the ultimate goal for fuzzing, and thus we also report the number of distinct bugs DeepREL finds.

# 6 RESULT ANALYSIS

## 6.1 RQ1: Effectiveness in API Coverage

In this RQ, we aim to study the effectiveness of DeepREL in terms of covering more APIs with API relations. Table 1 shows the number of DL library APIs covered by DeepREL and state-of-the-art FreeFuzz. Column "#Total" presents the total number of APIs in DL libraries, while Column "Improvement" presents the improvement of DeepREL over FreeFuzz. In the fuzzing stage, DeepREL covers 2973 APIs, which is a huge improvement (157%) over FreeFuzz that covers only 1158 APIs. For example, there are totally 1592 PyTorch APIs, and DeepREL can cover 1071 APIs, 128% more than FreeFuzz. A large number of APIs are not covered by FreeFuzz because they are less frequently used and not covered by any of the three sources of FreeFuzz. Leveraging API relations, DeepREL can successfully invoke these APIs with their relational APIs' inputs. The huge API coverage improvement demonstrates the potential of DeepREL.

**Table 3: Source distribution of inferred API pairs**

| | Src \ Tgt | Seed | New |
|---|---|---|---|
| **PyTorch** | **Seed** | 902 | 544 |
| | **New** | 598 | 2246 |
| **TensorFlow** | **Seed** | 777 | 1633 |
| | **New** | 830 | 5568 |

Table 2 further shows the number of verified API pairs detected by DeepREL. Columns "Equivalence$_{value}$" and "Equivalence$_{status}$" present the number of value-equivalent and status-equivalent API pairs accepted by the API Match Verifier respectively. API Match Verifier accepts 13098 API pairs in total, showing that such API relations are common in DL libraries. On PyTorch, DeepREL accepts more status-equivalent API pairs (2933) than value-equivalent (1357). The reason is that the latter relation is stricter than the former, and status-equivalent API pairs are more common. For example, in term of splitting a tensor, PyTorch provides a set of APIs: `torch.split`, `torch.tensor_split`, `torch.vsplit` (splits the tensor vertically), and `torch.dsplit` (splits the tensor depthwise). It is worth noting that TensorFlow has much more APIs than PyTorch, and DeepREL detects more value-equivalent API pairs than status-equivalent ones on TensorFlow. This is because TensorFlow contains lots of APIs for compatibility and low level access operations and thus has higher functional overlap: (1) The `tf.compat` module [8] contains 2579 redundant APIs to support forwards and backwards compatibility across TensorFlow versions (e.g., `v1` and `v2`). For example, `tf.compat.v1.layers.conv2d` is an alias for `tf.layers.conv2d`, and it allows user to use the `conv2d` layer with TensorFlow v1 behaviour in TensorFlow v2; (2) The `tf.raw_ops` module [9] contains 1339 low level APIs to provide direct access to all TensorFlow ops. For example, `tf.raw_ops.Pad` adds padding to tensors and is a low level API compared to the high-level API `tf.pad` with the same functionality.

Table 3 further presents a detailed distribution of the API pairs inferred by DeepREL based on whether the source/target APIs involve newly covered APIs. Column "Src" and Row "Tgt" present the categorization of the source and target APIs, respectively; Columns/Rows "Seed" and "New" indicate whether an API is from "seed APIs" covered by FreeFuzz or is newly covered by DeepREL. Out of all the 13098 API pairs verified by DeepREL, 1679 (902 + 777) only involve APIs covered by the original FreeFuzz, and all the remaining 11419 pairs involve newly covered APIs, which shows the importance of leveraging API relations to cover more APIs.

We also conduct a manual study to investigate why there are so many value-equivalent API pairs. Note that we do not look into status-equivalent API pairs because they are more intuitive (e.g., many APIs may share similar input parameter types and/or output behaviors). Since the number of verified value-equivalent API Pairs is huge, we select the set of equivalent API pairs explicitly specified in the documentations for our study. We mine the documentations for all 7973 PyTorch and TensorFlow APIs to extract API pairs when one API explicitly reference another API in the API documentation. In this way, we automatically extract 2942 API pairs and we manually categorize them in terms of why such relational APIs exist. 2692 out of 2942 API pairs are value-equivalent pairs. Note 1828 of them are backward compatibility pairs, and are unique to TensorFlow. Thus, we group the remaining pairs into 5 main reason categories as shown in Table 4. Ease of programming is the main reason for Equivalence$_{value}$ API pairs in both DL libraries. For example, `torch.det` is an alias for `torch.linalg.det`, and users can use the two symbols interchangeably. Meanwhile, Deprecation is one of the minor reasons. It is worth noting that we include the deprecated APIs in our study since they are still in the code base and can help cover more new APIs as well as find potential consistency bugs. Table 4 also provides examples to demonstrate each reason, where Column "Example ($S$, $T$)" refers to the (source API, target API) pair. The last column explains the difference between the relational APIs. For the Performance example, `tf.stack` and `tf.parallel_stack` are equivalent APIs which pile a list of tensor up. `parallel_stack` is more efficient than `stack` as the documentation of `tf.parallel_stack` says "`parallel_stack` *will copy pieces of*

*the input into the output as they become available, in some situations this can provide a performance benefit.*" [7].

## 6.2 RQ2: False Positives

False Positive Rate (FPR) is a common metric to evaluate the effectiveness of fuzz testing. Table 5 shows the FPR of DeepREL on the DL libraries. We analyze all inconsistencies reported by the API Fuzzer. Column "All" presents the total number of inconsistencies detected by API Fuzzer. "TP" (True Positive) means the number of true inconsistencies, and "FP" (False Positive) means the number of false alarms (e.g., inconsistencies due to incorrectly inferred matched API pairs). We separately report the statistics of the Equivalence$_{value}$ and Equivalence$_{status}$ oracles, while the "Overall" statistics report the merged results.

The FPR of DeepREL is only 30.72%, which implies that our API relation detection and verification techniques are effective. The false positives mainly originate from incorrect API relation verification. The API Match Verifier leverages valid inputs of the source API to decide whether an API relation candidate is correct or not. However, the valid inputs come from the FreeFuzz database and are not complete. As a result, the API Match Verifier can mistakenly accept a wrong API relation when the set of inputs does not cover certain part of the possible input space and thus is insufficient to distinguish the different behaviors of the API relations. We can also observe that the FPR of Equivalence$_{value}$ is lower than Equivalence$_{status}$. This is because that Equivalence$_{value}$ is stricter than Equivalence$_{status}$, which makes it easier for the API Match Verifier to reject wrong API relations over verifying inputs.

## 6.3 RQ3: Impacts of Configuration

In this RQ we analyze how different configurations affect the performance of DeepREL, including API coverage, False Positive Rate (FPR), and the running time. We focus on two hyper-parameters, $K$ (the number of matched pairs for each source API, discussed in Section 4.1) and $I$ (the number of iterations, discussed in Section 4.5). The default values for $K$ and $I$ are both 10 in DeepREL. To figure out the impact of $K$, we run our experiments with different $K$ values of 5, 10, 15, 20. We also run DeepREL for up to 10 iterations and show the impact of different $I$ values from 1 to 10. Figures 8 and 9 show the results under different configurations. The $x$ axis shows the iterations, and the $y$ axis presents the number of covered APIs, FPR, and the running time. The results for different $K$ values are shown in different lines. We can observe that DeepREL will terminate (i.e., reaching fixed points) in at most 8 iterations on PyTorch, and on TensorFlow it either terminates in at most 8 iterations, or only covers 2 new APIs in the last (i.e. 10th) iteration.

The impact of $K$ is similar for PyTorch and TensorFlow. First, the number of covered APIs of $K = 10, 15, 20$ are close, all significantly higher than $K = 5$. Second, the FPR increases as $K$ increases. The reason is that less similar APIs can incur more false positives, indicating the effectiveness of our API similarity computation. Third, the running time is increasing approximately linearly with $K$. Therefore, it is reasonable to set $K$ as 10 in our default setting, as setting $K$ to be higher than 10 would bring marginal benefit in terms of API coverage, but degrade FPR and time cost.

For the impact of $I$, we have the following observations. First, as expected, the number of covered APIs increases at a lower and lower pace with the increase of $I$, and converges within 10 iterations. Second, the false positive rate generally increases with the iteration. The reason is that source APIs for later iterations are typically target APIs from earlier iterations, and may have less and less valid inputs (since they may fail on some inputs from the original source APIs) for verifying inferred API pairs. The only exception is that FPR drops when $I$ increases to 2 with $K = 5$ or 10 on PyTorch. We look into the data and observe that DeepREL is able to accidentally detect a large number of true positives in the $2^{nd}$ iteration (e.g., `torch.Tensor.*` APIs and their value-equivalent APIs `torch.*`). Third, the running time increases more dramatically at early iterations and grows slowly in later iterations, which is consistent with the growth of the number of covered APIs.

We can also observe that the total running time of the default DeepREL is 12.8h for PyTorch and 26.3h for TensorFlow. Such cost is actually quite common for fuzzing techniques, e.g., various fuzzing techniques have been applied for 24h or even more [18, 40, 58, 66, 78] (including the recent LEMON work [64]).

## 6.4 RQ4: Bugs Detected

Table 6 presents the summary of real-world bugs detected by DeepREL for the two studied DL libraries. Column "Total" shows the total number of bugs detected by DeepREL, and Columns "Value" and "Status" show the number of bugs detected with the Equivalence$_{value}$ and Equivalence$_{status}$ oracle respectively. We also present the number of bugs rejected by developers (Column "Rejected"), confirmed as previously unknown (Column "Confirmed"), and the number of previously unknown bugs that have already been fixed (Column "Fixed"). We can observe that DeepREL is able to detect 162 bugs in total, with only 7 rejected by developers, 106 confirmed by developers as previously unknown bugs (29 already fixed), and all others pending. Among those 106 confirmed bugs, only 9 can be found by FreeFuzz, and none of them can be detected by CRADLE, AUDEE, or LEMON. Furthermore, we have also found 10 documentation bugs for PyTorch and 4 for TensorFlow during the experiment (these document bugs are not included in Table 6).

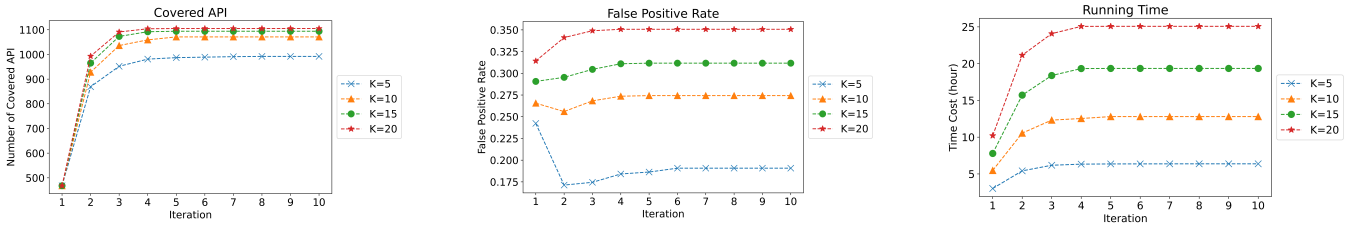
The bugs detected by DeepREL can also be categorized into single API bugs[2] and consistency bugs between relational API pairs. Table 7 shows the breakdown of the 106 confirmed bugs based on the two cases. For the 19 single API bugs detected by DeepREL, 9 are in the "seed APIs" (Column "Seed"), while 10 are bugs in the newly covered APIs (Column "New"), emphasizing the importance of leveraging API relations to cover more APIs. Meanwhile, all the remaining 87 ones are consistency bugs, demonstrating the effectiveness of using API relations as the oracle for DL library testing. More specifically, out of the 87 consistency bugs, 35 are inconsistencies between "seed APIs" (Column "Seed-only") while 52 involve newly covered APIs (Column "Others"), further indicating that covering new APIs contributes to consistency bug detection.

Notably, DeepREL is able to detect 23 high-priority bugs for PyTorch (note that TensorFlow is not discussed here as it does not

[2]The single API bugs are unexpected crashes caused by single buggy APIs. In Table 6, we categorize such bugs as "Value"/"Status" if they were detected when testing API pairs with the Equivalence$_{value}$/Equivalence$_{status}$ oracle (although they can be detected without such relational API oracles).

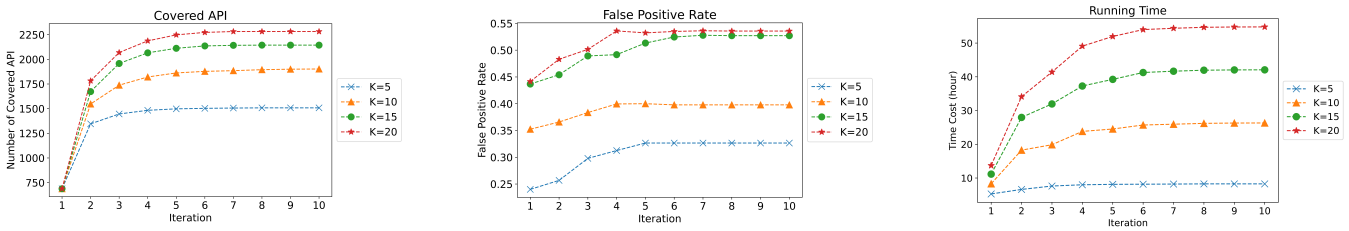
**Table 4: Classification of reasons for Equivalence$_{value}$ API pairs**

| Reason | # TF | % TF | # PT | % PT | Example ($S$, $T$) | Diff. between $S$ and $T$ |
|---|---|---|---|---|---|---|
| **Ease of programming** | 460 | 95.44% | 349 | 91.36% | `(torch.det, torch.linalg.det)` | $S$ is an alias for $T$ |
| **Performance** | 10 | 2.07% | 9 | 2.36% | `(tf.parallel_stack, tf.stack)` | $S$ uses parallelism for efficiency. |
| **Special cases** | 10 | 2.07% | 10 | 2.62% | `(tf.boolean_mask, tf.ragged.boolean_mask)` | $T$ extends $S$ to ragged tensors. |
| **Numerical stability** | 1 | 0.21% | 4 | 1.05% | `(torch.linalg.inv, torch.linalg.solve)` | $T$ is faster and more numerically stable. |
| **Deprecation** | 1 | 0.21% | 10 | 2.62% | `(torch.qr, torch.linalg.qr)` | $S$ is deprecated. |




**(a) Number of covered APIs** **(b) False positive rate** **(c) Running time**

**Figure 8: Analysis of hyper-parameter top-$K$ and iteration $I$ on PyTorch**




**(a) Number of covered APIs** **(b) False positive rate** **(c) Running time**

**Figure 9: Analysis of hyper-parameter top-$K$ and iteration $I$ on TensorFlow**

**Table 5: False positive rate of DeepREL**

| | Oracle | # All | TP | FP | FPR |
|---|---|---|---|---|---|
| **PyTorch** | **Equivalence$_{value}$** | 412 | 364 | 48 | 11.65% |
| | **Equivalence$_{status}$** | 853 | 554 | 299 | 35.05% |
| | **Overall** | 1265 | 918 | 347 | 27.43% |
| **TensorFlow** | **Equivalence$_{value}$** | 97 | 68 | 29 | 29.90% |
| | **Equivalence$_{status}$** | 363 | 209 | 154 | 42.42% |
| | **Overall** | 460 | 277 | 183 | 39.78% |
| **Total** | **Equivalence$_{value}$** | 509 | 432 | 77 | 15.13% |
| | **Equivalence$_{status}$** | 1216 | 763 | 453 | 37.25% |
| | **Overall** | 1725 | 1195 | 530 | 30.72% |

have such labels). These bugs are marked as "high priority" because they are critical and require urgent resolution. The reported "high priority" bugs are highly valued by PyTorch developers and have raised heated discussions. During the three months from our first bug issue (November 23, 2021) to March 1, 2022, there have been a total of 170 high-priority issues in the entire PyTorch issue-tracking system [11], of which we contributed 23 (13.5%), emphasizing the effectiveness of DeepREL.

We next present some example bugs detected by DeepREL:

**Out-of-Bounds Read (Equivalence$_{value}$, ✓)** The bug shown in Figure 10 is detected for API pair `torch.kthvalue(tensor, k)` and `tensor.kthvalue(k)` via Equivalence$_{value}$. The latter API is one of the methods from `torch.Tensor` [6], which is the fundamental class of PyTorch. Given the same tensor (`input`) and argument (`k`), the returned results (`result1` and `result2`) are not equal (i.e., assertion fails on Line 5). After debugging, we find that the returned results

```
1 input = torch.tensor([0,1,2,3,4])
2 k = 6
3 result1 = torch.kthvalue(input,k)
4 result2 = input.kthvalue(k)
5 torch.testing.assert_close(result1, result2) # fail
```

**Figure 10: Out-of-Bound Read in `torch.kthvalue`**

can actually be different across runs, indicating that it reads values from memory locations outside of user-controlled data! This bug is a silent error, and has a severe security implication: without proper range checking of `k`, users may be able to read data outside of the allocated memory bounds (i.e., out-of-bound read). This bug is labeled as "high priority", and developers have fixed it immediately.

**Inconsistent Check (Equivalence$_{status}$, ✓)** Figure 11 shows a bug for `torch.nn.AdaptiveAvgPool3d` and `torch.nn.AdaptiveMaxPool3d` (found via Equivalence$_{status}$). With exactly the same input tensor (`tensor`) and argument (`output_size`), `AdaptiveAvgPool3d` runs without exception (Line 4) while `AdaptiveMaxPool3d` throws `RuntimeError: Trying to create tensor with negative dimension` (Line 6). After inspection, we find that `AdaptiveAvgPool3d` lacks checking for dimensions with negative integers. This can be disastrous since users may inadvertently introduce a bug into their model, but no warning/exception is raised. This bug has also been fixed.

**Wrong Computation (Equivalence$_{value}$, ×)** The bug shown in Figure 12 is detected by Equivalence$_{value}$ oracle for API pair `torch.std_mean` and `torch.mean`. Both of them could compute the mean of all elements in the `input` tensor. However, DeepREL has

**Table 6: Summary of detected bugs**

| | Total | | | Rejected | | | Confirmed | | | Fixed | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Total** | Value | Status | **Total** | Value | Status | **Total** | Value | Status | **Total** | Value | Status |
| PyTorch | **121** | 76 | 45 | **6** | 3 | 3 | **71** | 33 | 38 | **17** | 9 | 8 |
| TensorFlow | **41** | 22 | 19 | **1** | 0 | 1 | **35** | 22 | 13 | **12** | 3 | 9 |
| Total | **162** | 98 | 64 | **7** | 3 | 4 | **106** | 55 | 51 | **29** | 12 | 17 |

**Table 7: Source distribution of confirmed bugs**

| | | Single API bugs | | | Relational API bugs | | |
|---|---|---|---|---|---|---|---|
| | **Total** | **Total** | **Seed** | **New** | **Total** | **Seed-only** | **Others** |
| **PT** | 71 | 18 | 9 | 9 | 53 | 26 | 27 |
| **TF** | 35 | 1 | 0 | 1 | 34 | 9 | 25 |
| **Total** | 106 | 19 | 9 | 10 | 87 | 35 | 52 |

```
output_size = [-36, 0, 0]
tensor = torch.rand([4, 4, 128, 2048, 4])
layer1 = torch.nn.AdaptiveAvgPool3d(output_size)
result1 = layer1(tensor) # no exception
layer2 = torch.nn.AdaptiveMaxPool3d(output_size)
result2 = layer2(tensor) # RuntimeError
```

**Figure 11: Inconsistent check for `torch.nn.AdaptiveAvgPool3d`**

```
1 input = torch.tensor([[0.5786, 0.1719, 0.3760, 0.2939, 0.3984],
2         [0.5361, 0.7104, 0.8765, 0.0903, 0.0483]], dtype=torch.float16)
3 result1 = torch.std_mean(input) # 0.4080
4 result2 = torch.mean(input) # 0.4082
5 torch.testing.assert_close(result1, result2) # fail
```

**Figure 12: Inconsistency for `torch.std_mean` and `torch.mean`**

found that for certain input tensors, `torch.std_mean` has different returned value with `torch.mean` (i.e., assertion fails on Line 5). Although we believe it is a bug, the developers said these two APIs are not expected to output the same mean and rejected this bug report. Note that the other 6 rejected bugs are similar (there are indeed inconsistencies, but developers feel unnecessary to fix).

## 6.5 Threats to validity

The threats to internal validity mainly lie in the correctness of the DeepREL implementation. To reduce such threats, the first three authors have performed extensive testing and code review of DeepREL. Moreover, we have released our code/scripts in our project website for public review [12]. The threats to external validity mainly lie in the evaluation benchmarks used. To demonstrate the generalizability of DeepREL, we have evaluated DeepREL on two widely used DL libraries, TensorFlow and PyTorch. Last but not least, the threats to construct validity mainly lie in the metrics used. To reduce such threats, we adopt the number of covered APIs and detected bugs, following prior work on DL library testing [53, 64, 65]. Moreover, we have also included false positive analysis.

# 7 RELATED WORK

We have talked about related work on DL library testing in Section 2. Therefore, in this section, we mainly focus on existing work targeting API relations for other software systems. API mappings refer to the process of finding equivalent APIs between different libraries or programming languages. Prior work [48, 49] proposed to train neural networks to mine API mappings by learning transformations between the vector spaces of APIs from different languages (e.g., Java and C#). Another work [19] leveraged unsupervised domain adaptation approach to automatically construct and align vector spaces for identifying API mappings with much less human efforts. Our work is different from all such work in that DeepREL focuses on fuzzing while aforementioned work targets code migration. Also, DeepREL utilizes not only equivalent APIs, but also other relational APIs for fuzzing. Moreover, DeepREL obtains relational APIs purely from documentation, and then verifies the API relations dynamically; thus, DeepREL does not need extensive API usages or training data of existing API pairs (which can be hard to obtain).

Function synonyms [22] are functions that play a similar (*not* necessarily semantically equivalent) role in code. UC-KLEE [55] leverages symbolic execution to check two different implementations/versions of the same function in C open source libraries. Func2vec [22] is a technique for finding function synonyms in Linux file systems and drivers via learning function embeddings. Func2vec trains a neural network on sentences generated using random walks of the interprocedural control-flow graph of the program, and it is applied to improve the quality of error handling specifications for Linux code. More recently, FPDiff [63] aims to automatically identify synonymous functions across multiple numerical libraries, and performs differential testing to detect discrepancies of these function synonyms. While FPDiff is closely related, it mainly considers synonymous APIs with same functionalities and argument lists across different libraries and is typically applied when there exist libraries with close design; in contrast, DeepREL is more general – it considers arbitrary relational APIs with value/status equivalence within *any given* library. Also, FPDiff can only target numerical library APIs taking `double` or `int` variables as input, whereas the input of APIs in DL libraries can be much more complex (e.g., tensors of different dimensions/types, objects, strings, etc.). Moreover, FPDiff is a purely dynamic technique that finds synonymous functions via running them on an *elementary* set of `double/int` values, while it is impossible to find such an universal elementary input set for DL library APIs. Lastly, our study has found that DL library documentations contain various valuable information and shown for the first time that there can be plenty of equivalent/similar APIs within a given DL library, which can substantially help with fuzzing DL libraries (and beyond).

Throughout this paper, we have viewed each API as a test object, and thus our technique of fuzzing relational API pairs falls into the differential testing category (following FPDiff [63]). Meanwhile, if we treat the entire systems under test (e.g., TensorFlow or PyTorch) as test objects, this work can also be viewed as an instance/extension of the metamorphic oracle generation work, which aims to automatically infer metamorphic relations as test oracle [17, 27, 34, 43, 62, 73, 81, 82]. For example, SBES [27, 43] synthesizes sequences of method invocations that are equivalent to

a target method based on the observed dynamic program behaviors. The recent state-of-the-art technique MeMo [17] mainly targets mature open-source Java projects and automatically derives metamorphic relations from natural language specifications. Meanwhile, MeMo can only infer metamorphic relations that are well documented and lacks invocation synthesizer or API relation verifier; thus the effectiveness of MeMo strictly depends on documentation completeness, correctness, and preciseness. In fact, very few metamorphic relations in DL libraries are explicitly described with informative sentences (with code templates) and can be directly translated into valid oracles. We substituted the first three phases of DeepREL with MeMo, and found that it can infer only 17/28 API relations for TensorFlow/PyTorch (compared to 8808/4290 relations found by DeepREL), and can at most detect 8 (including 6 found by MeMo and 2 found by FreeFuzz on the APIs involved in the MeMo inferred relations) of the 106 confirmed bugs detected by DeepREL.

## 8 CONCLUSION

We have introduced DeepREL, the first fully automated end-to-end approach to fuzzing DL libraries via inferring relational APIs. DeepREL can "borrow" test inputs from any API to test its relational APIs, and can leverage relational APIs as reference implementations for performing differential testing. The extensive study of DeepREL on PyTorch and TensorFlow shows that DeepREL is able to detect 162 bugs in total, with 106 already confirmed by the developers as previously unknown bugs. Notably, DeepREL has detected 13.5% of the high-priority bugs for the entire PyTorch issue-tracking system in a three-month period. Also, besides the 162 code bugs, we were also able to detect 14 documentation bugs (all confirmed).

## REFERENCES


[1] Keras, 2015. https://keras.io.

[2] bs4, 2021. https://pypi.org/project/beautifulsoup4/.

[3] Definition of torch.broadcast_shapes from Pytorch official documentation, 2021. https://pytorch.org/docs/stable/generated/torch.broadcast_shapes.html.

[4] News, 2021. https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash.

[5] Topological Sorting, 2021. https://en.wikipedia.org/wiki/Topological_sorting.

[6] Class torch.Tensor from PyTorch official documentation, 2022. https://pytorch.org/docs/stable/tensors.html#torch.Tensor.

[7] Definition of tf.parallel_stack from TensorFlow official documentation, 2022. https://www.tensorflow.org/api_docs/python/tf/parallel_stack.

[8] Module tf.compat from TensorFlow official documentation, 2022. https://www.tensorflow.org/api_docs/python/tf/compat.

[9] Module tf.raw_ops from TensorFlow official documentation, 2022. https://www.tensorflow.org/api_docs/python/tf/raw_ops.

[10] munkres, 2022. https://pypi.org/project/munkres/.

[11] PyTorch Repository, 2022. https://github.com/pytorch/pytorch.

[12] RelFuzz Repository, 2022. https://github.com/RelFuzzer/RelFuzz.

[13] SentenceTransformer, 2022. https://www.sbert.net/.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[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] G. Amir, H. Wu, C. Barrett, and G. Katz. An smt-based approach for verifying binarized neural networks. *arXiv preprint arXiv:2011.02948*, 2020.

[17] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation. *Journal of Systems and Software*, 181:111041, 2021.

[18] 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.

[19] N. Bui. Towards zero knowledge learning for cross language api mappings. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 123–125. IEEE, 2019.

[20] 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.

[21] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.

[22] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

[23] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[24] A. Eitel, J. T. Springenberg, L. Spinello, M. Riedmiller, and W. Burgard. Multimodal deep learning for robust rgb-d object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 681–687. IEEE, 2015.

[25] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[26] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 213–224, 2016.

[27] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 366–376, 2014.

[28] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[29] 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.

[30] X. Gu, H. Zhang, and S. Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[31] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–498. IEEE, 2020.

[32] 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.

[33] 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.

[34] U. Kanewala. Techniques for automatic detection of metamorphic relations. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 237–238. IEEE, 2014.

[35] 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.

[36] I. Lenz, H. Lee, and A. Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

[37] 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.

[38] Y. Li, S. Wang, and T. N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[39] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, and M. J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.

[40] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[41] 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.

[42] 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.

[43] A. Mattavelli, A. Goffi, and A. Gorla. Synthesis of equivalent method calls in guava. In *International Symposium on Search Based Software Engineering*, pages 248–254. Springer, 2015.

[44] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936, 2019.

[45] S. Min, B. Lee, and S. Yoon. Deep learning in bioinformatics. *Briefings in bioinformatics*, 18(5):851–869, 2017.

[46] 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.

[47] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[48] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mapping api elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758. IEEE, 2016.

[49] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] M. F. Porter. Snowball: A language for stemming algorithms, 2001.

[55] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *International Conference on Computer Aided Verification*, pages 669–685. Springer, 2011.

[56] D. Ravì, C. Wong, F. Deligianni, M. Berthelot, J. Andreu-Perez, B. Lo, and G.-Z. Yang. Deep learning for health informatics. *IEEE journal of biomedical and health informatics*, 21(1):4–21, 2016.

[57] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019.

[58] 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.

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

[60] 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.

[61] 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.

[62] J. Troya, S. Segura, and A. Ruiz-Cortés. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software*, 136:188–208, 2018.

[63] J. Vanover, X. Deng, and C. Rubio-González. Discovering discrepancies in numerical libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 488–501, 2020.

[64] 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.

[65] A. Wei, Y. Deng, C. Yang, and L. Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 995–1007, 2022.

[66] 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.

[67] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[68] Wikipedia contributors. Maximum weight matching — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Maximum_weight_matching&oldid=1033788761, 2021.

[69] Wikipedia contributors. Wikipedia for backpropagation, 2021. https://en.wikipedia.org/wiki/Backpropagation.

[70] Wikipedia contributors. Wikipedia for Supervised Learning, 2021. https://en.wikipedia.org/wiki/Supervised_learning.

[71] Wikipedia contributors. Levenshtein distance — Wikipedia, the free encyclopedia, 2022. [Online; accessed 12-March-2022].

[72] Wikipedia contributors. Tf–idf — Wikipedia, the free encyclopedia, 2022. [Online; accessed 6-March-2022].

[73] Z. Xiang, H. Wu, and F. Yu. A genetic algorithm-based approach for composite metamorphic relations construction. *Information*, 10(12):392, 2019.

[74] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022. To appear.

[75] 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.

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

[77] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 435–450, 2021.

[78] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*, pages 769–786. IEEE, 2019.

[79] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022. To appear.

[80] 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.

[81] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato. Automatic discovery and cleansing of numerical metamorphic relations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245. IEEE, 2019.

[82] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 701–712, 2014.

[83] 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.

[84] 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.

[85] S. Zheng, R. Chen, Y. Jin, A. Wei, B. Wu, X. Li, S. Yan, and Y. Liang. Neoflow: A flexible framework for enabling efficient compilation for high performance dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[86] 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.