EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries

Jiannan Wang
Purdue University
West Lafayette, USA
wang4524@purdue.edu

Thibaud Lutellier
University of Waterloo
Waterloo, Canada
tlutelli@uwaterloo.ca

Shangshu Qian
Purdue University
West Lafayette, USA
shangshu@purdue.edu

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

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

ABSTRACT

Testing deep learning (DL) software is crucial and challenging. Recent approaches use differential testing to cross-check pairs of implementations of the same functionality across different libraries. Such approaches require two DL libraries implementing the same functionality, which is often unavailable. In addition, they rely on a high-level library, Keras, that implements missing functionality in all supported DL libraries, which is prohibitively expensive and thus no longer maintained.

To address this issue, we propose *EAGLE*, a new technique that uses differential testing in a different dimension, by using equivalent graphs to test *a single DL implementation* (e.g., a single DL library). Equivalent graphs use different Application Programming Interfaces (APIs), data types, or optimizations to achieve the same functionality. The rationale is that two equivalent graphs executed on a single DL implementation should produce identical output given the same input. Specifically, we design 16 new DL equivalence rules and propose a technique, EAGLE, that (1) uses these equivalence rules to build concrete pairs of equivalent graphs and (2) cross-checks the output of these equivalent graphs to detect inconsistency bugs in a DL library.

Our evaluation on two widely-used DL libraries, i.e., TensorFlow and PyTorch, shows that EAGLE detects 25 bugs (18 in TensorFlow and 7 in PyTorch), including 13 previously unknown bugs.

CCS CONCEPTS

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

KEYWORDS

software testing, deep learning, differential testing, graph equivalence

1 INTRODUCTION

Testing DL systems is crucial because an increasing number of DL systems, e.g., self-driving cars and cancer detection, have been deployed. Bugs in DL systems cause severe consequences; for example, when a self-driving system incorrectly responds to a traffic sign, it causes severe personal injury and economic damage [8].
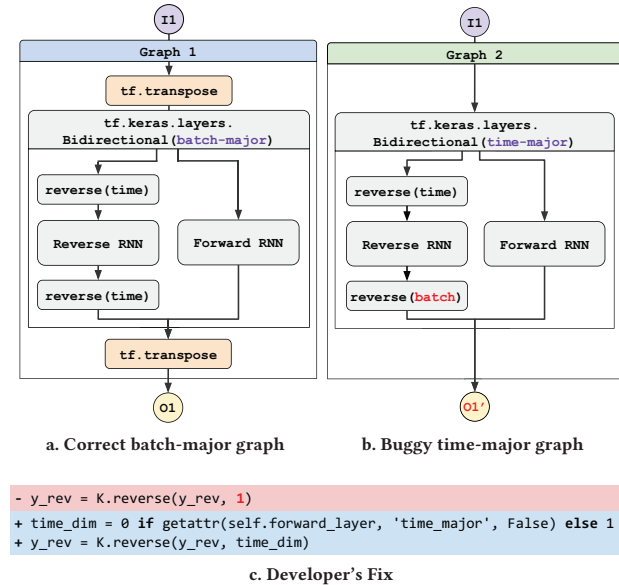
When DL software fails to implement a model faithfully, e.g., due to a bug in the software, the output from the software can be wrong even if the model is correct [31]. Here DL software includes infrastructure code that performs core neural network computations and application code that loads model weights. Thus, in addition to testing DL models [9, 22, 26, 36–38, 40, 46, 49], there is a high demand for testing DL software [14, 28, 29, 31, 42, 43, 45, 47].

Existing techniques such as CRADLE [31] and Audee [14] test a pair of DL libraries to cross-check the two implementations of the same functionality to detect inconsistency bugs. These differential testing techniques require at least two implementations in different DL libraries, which is often unavailable for DL software. For example, one could implement a new DL algorithm in one library, e.g., TensorFlow [1], which does not have a counterpart in another library (e.g., CNTK [33]). Since only one single implementation exists, existing cross-library testing techniques cannot test it.

In addition, differential testing on two libraries [14, 31] requires a high-level library such as Keras [4] to switch across DL libraries such as TensorFlow and CNTK. Such a high-level library is hard to develop and maintain because it essentially reimplements functionalities that are only available in one library in all other supported libraries. This is one of the main reasons why Keras stopped supporting different DL libraries [18]. Without such a high-level library, it would be prohibitively expensive to cross-check DL libraries because one would need to create separate, complex DL implementations for other DL libraries.

To address these challenges, we propose to leverage differential testing in a different dimension: our tool, *EAGLE*, uses *equivalent graphs* to test a single DL implementation. For example, the classification output should be identical if a DL implementation uses

a. Correct batch-major graph **b. Buggy time-major graph**

```
- y_rev = K.reverse(y_rev, 1)
+ time_dim = 0 if getattr(self.forward_layer, 'time_major', False) else 1
+ y_rev = K.reverse(y_rev, time_dim)
```

c. Developer's Fix

Figure 1: Equivalent graphs that we designed to detect a real bug in TensorFlow. Red background indicates the buggy line. $I1$ is the tensor input. $O1$ and $O1'$ are output that is expected to be identical. The bug causes $O1 \neq O1'$.

two different but equivalent Recurrent Neural Networks (RNN) to perform a classification task. We define *equivalent graphs* as *computational graphs that achieve the same functionality, which should produce identical output given the same input.* The equivalence is achieved with different APIs, data types, optimizations, etc.

For example, an optimized way of representing mostly-empty tensors of DL models is using sparse tensors. One can generate two equivalent computation graphs: the first taking a dense tensor as input and the second taking a sparse tensor as input. While these two graphs may invoke different API functions, they are equivalent, i.e., they should produce the same output given the same input represented as a dense tensor in the first graph, or a sparse tensor in the second. When the API functions contain bugs, the output may be different. EAGLE detects seven bugs related to sparse tensors in TensorFlow and PyTorch using equivalent graphs.

1.1 Our Approach

A motivating example: Figure 1 shows a real bug in TensorFlow 2.1 detected by EAGLE using two equivalent graphs. The equivalence rule used to generate the two equivalent graphs in Figures 1a and 1b is inspired by RNN functions that accept two input formats. A common format is [batch, time] (called *batch-major*), which is the usual input format developers use. The other format is [time, batch] (called *time-major*). The time-major format better fits RNN computations because RNNs compute batches step-by-step, and similar steps from different sequences are represented contiguously in flattened time-major arrays, thus reducing training time. For example, given the following two batches of three words (i.e., three

time steps) [I like dogs] and [I eat apples], the input can be fed to the RNN in batch-major format (i.e., $\begin{bmatrix} I & like & dogs \\ I & eat & apples \end{bmatrix}$) or time-major format (i.e., $\begin{bmatrix} I & I \\ like & eat \\ dogs & apples \end{bmatrix}$). The first matrix is the transpose of the second matrix. Developers use an argument (e.g., parameter time_major True or False in TensorFlow) of RNN functions to specify input's format. By transposing the correct dimension, one can transform a time-major input matrix to a batch-major input matrix.

Therefore, leveraging the time-major/batch-major and transpose properties, we create two equivalent graphs. The graph in Figure 1a first transposes the time-major input tensor $I1$ to batch-major, feeds it to a batch-major RNN (tf.keras.layers.Bidirectional in this example), then transposes the output back to time-major. The graph in Figure 1b directly feeds the original time-major input to the time-major RNN to produce a time-major output. If the RNN API implementation is correct, these two equivalent graphs should generate the same output given the same time-major input.

Figure 1b shows a real bug in the TensorFlow API function tf.keras.layers.Bidirectional (which implements bidirectional RNNs) and how the bug causes an inconsistency: the same function tf.keras.layers.Bidirectional generates different output $O1$ and $O1'$ given the same input $I1$ (e.g., a tensor representation of [I like dogs]) on two equivalent graphs. The bug is in red in Graph 2 (Figure 1b) since the function reverse should be performed on the time dimension instead of the batch dimension. The bidirectional RNN consists of two independent RNNs: a forward RNN and a reverse RNN. The forward RNN processes the input in the normal order, and the reverse RNN in the reverse order (e.g., "I like dogs" becomes "dogs like I"). Since the output of the reverse RNN is not in the correct order, it needs to be reversed. The API's batch-major mode (Figure 1a) correctly uses the reverse function on the time dimension, but its time-major mode (Figure 1b) incorrectly reverses the batch dimension instead of the time dimension, i.e., reverse(batch) (in red) is incorrect and should be reverse(time), resulting in incorrect output $O1'$.

It is challenging to detect this bug without EAGLE because without Graph 1 in Figure 1a, one may not know $O1'$ in Graph 2 in Figure 1b is the wrong output for input $I1$. The reason is that it is hard to know the expected output $O1$ given input $I1$ since the DL calculation (e.g., reverse RNN) is complex [31]. Our equivalent graph approach addresses this challenge by comparing the output from two equivalent graphs to identify inconsistencies to detect software bugs.

Figure 1c shows the fix provided by TensorFlow developers, who fixed the bug by setting the appropriate dimension to reverse according to the input format, with the buggy line in red background and the fixed line in blue background.

Such graph equivalence on time-major and batch-major is general as most DL libraries, including TensorFlow and PyTorch, use such representation. We apply EAGLE to test 13 RNN functions in TensorFlow and PyTorch and detect that *all* bidirectional RNNs in TensorFlow incorrectly implement the time-major functionality.

This motivating example demonstrates how equivalent graphs enable the discovery of hard-to-find bugs in DL libraries. We present below the main steps of our approach.

Equivalence rule definition: The first step is to generate rules to build equivalent graphs. There are two main criteria for generating these rules. First, the rules should be generalizable to multiple APIs and DL libraries. Second, the rules should be non-trivial to detect real-world bugs. To cover as many libraries as possible, we carefully inspect API documentation from TensorFlow and PyTorch libraries. In total, we design a list of 16 new equivalence rules that covers 1,427 APIs of these two DL libraries.

Equivalent graph construction: Once we have a set of equivalence rules, we concretize these general abstract rules into concrete graphs. Specifically, we test a concrete DL function with specific configurations (e.g., weights) and input. For example, the rule presented in Figure 1 applies to any RNN function (e.g., RNN, LSTM, GRU, and biLSTM). This results in 10 pairs of TensorFlow equivalent graphs, each is tested with 400 sets of (input, configuration). We follow previous work [43] to generate valid input based on constraints automatically extracted from the API documentation.

Bug detection: We compare the output from a pair of concrete equivalent graphs to detect inconsistency bugs.

1.2 Contributions

In this paper, we make the following contribution:

- We design 16 new *equivalence rules* to create equivalent graphs to test DL libraries. These rules cover six categories of DL graph equivalence, i.e., optimization, API redundancy, data structure equivalence, data format equivalence, inverse equivalence, and model evaluation equivalence.
- We propose a novel idea of using equivalent graphs to detect bugs and implement this idea as a new testing technique—EAGLE, that generates equivalent graphs and detects bugs in DL libraries.
- We evaluate EAGLE on five of the latest versions of the most popular DL libraries (TensorFlow and PyTorch). Using the 16 rules, EAGLE generates 6,861 pairs of equivalent graphs and detects 25 bugs (18 in TensorFlow and 7 in PyTorch), including 13 previously unknown bugs.

Availability: Data is available in our GitHub repository[1].

The rest of the paper is organized as follows. Section 2 presents the definition of key concepts such as graphs, inputs, and configurations. Section 3 describes the equivalence rules and EAGLE's implementation. Section 4 describes our experimental setup. In Section 5, we evaluate EAGLE on two popular DL libraries, describe some bugs that EAGLE detects, compare EAGLE to state-of-the-art DL testing techniques, and present its execution time. Sections 6 and 7 respectively describe threats to validity and related work. Finally, Section 8 concludes the paper.

2 DEFINITION AND TERMINOLOGY

A *graph* in this paper represents a computational graph in which the nodes are operations performed on variables.

A set of (input, configuration) is required to compute a *graph* and generate an output. The *input* ($I1$ in Figure 1) is the object,

---

[1]https://github.com/lin-tan/eagle

often one or several tensors, on which the computation is done. We call *configuration* all the other arguments necessary to perform the computation (e.g., weights, number of neurons, etc.). For simplicity, we only list the input I in the equivalence rules (Table 1), but the configuration of the two equivalent graphs is assumed to be identical, except when explicitly described in the rule. For example, for the batch-major/time-major rule presented in Figure 1, the two graphs have identical configurations (weights and other arguments), except for the `time_major` argument, which is `False` in Graph 1, and `True` in Graph 2. Since it is the only difference, it is the only configuration explicitly described in Table 1 for this rule.

3 APPROACH

Finding bugs, especially non-crash bugs, in DL libraries is challenging because it is difficult to know the expected output, given that DL computations are complex. We cannot use the ground truth as the expected output of DL software since DL models are not 100% accurate [31]. When a model makes a mistake on input I, the expected output O of the software is different from the ground truth output O^0. EAGLE uses differential testing to address this challenge to find non-crash bugs.

Figure 2 presents the overview of our work, which consists of three main steps. First, we define generalizable rules for creating equivalent graphs (Step 1 in Figure 2). Second, for each rule, we obtain applicable APIs by checking DL API's documentation, and build pairs of concrete equivalent graphs (Step 2). Finally, we execute the two equivalent graphs by feeding them fuzzed input [43] and compare their output (for example $O1$ and $O1'$ in Figure 2) to detect inconsistency bugs (Step 3).

The rest of the approach section describes the equivalence rules (Section 3.1), the equivalent graph construction (Section 3.2), and the bug detection process (Section 3.3).

3.1 Equivalence Rules

The first step is to create rules to build equivalent graphs. Recall that *equivalent graphs* are computational graphs that achieve the same functionality, which should produce identical output given the same input. In practice, if the output difference is below a threshold t, we also consider the outputs identical. Such a threshold is needed because DL computations are mostly performed on floating numbers, and equivalent floating-point computations often result in slightly different outputs.

To create equivalence rules that are more likely to find real bugs in DL libraries, we examine the following two sources:

(1) API documentation: The API documentation of neural network functions provides us with information about their implementation. Sometimes, the description of several APIs provides connections among these APIs that help us create equivalence rules. For example, by reading the description of the function `tf.keras.layers.DepthwiseConv2D`, we found that this function could be implemented by multiple invocations of the function `tf.keras.layers.Conv2D`, each of which is performed over a single channel of the input to `tf.keras.layers.DepthwiseConv2D`. This implementation using `tf.keras.layers.Conv2D` is different from the implementation of `tf.keras.layers.DepthwiseConv2D` in TensorFlow. Although `tf.keras.layers.DepthwiseConv2D`
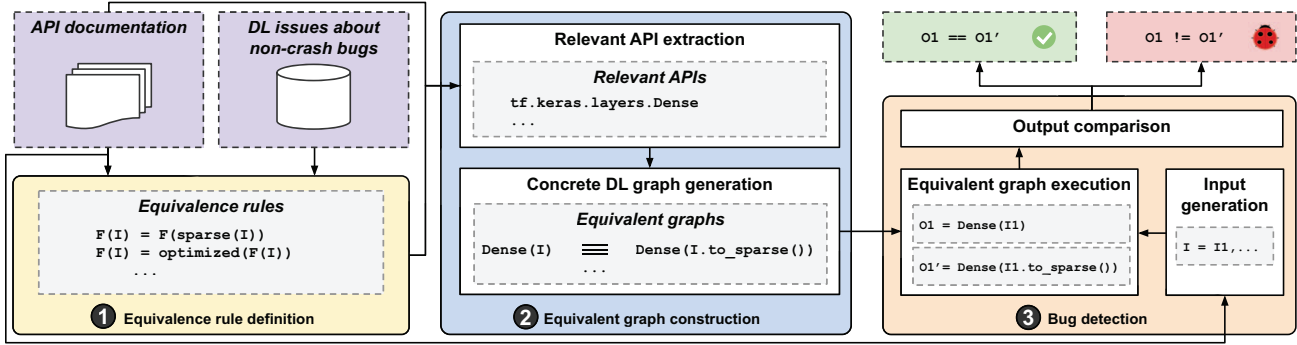
Figure 2: Overview of EAGLE

can be implemented using `tf.keras.layers.Conv2D`, TensorFlow chooses a more efficient implementation and computes the depthwise convolution directly without splitting it by the channels. Therefore, we have two different but equivalent implementations of `tf.keras.layers.DepthwiseConv2D` and we can use them to build equivalent graphs to detect bugs.

(2) Non-crash bugs in DL libraries: Similar to prior work [17], we study non-crash bugs in DL libraries to summarize common bug patterns and equivalence rules that can potentially detect those bugs. We first manually investigate GitHub issues related to non-crash bugs in TensorFlow and PyTorch's repositories. Then we reproduce those bugs and build a pair of equivalent graphs to detect each bug for the particular buggy API described in the issue. We then convert the graphs to a general equivalence rule by abstracting the inputs, API functions, and configurations (e.g., metrics and optimizations used).

Designing Equivalence Rules: We first create a concrete equivalence rule for a specific API for which we read the documentation or that contains a known bug. Then, we generalize the rule by abstracting the inputs, API functions, and configurations (e.g., metrics and optimization used). For example, the concrete rule used for the API function `tf.keras.layers.Bidirectional` in Figure 1 is $(\text{Bidirectional}(I1^T, ..., \text{return\_sequences}, ..., \text{batch-major}))^T$ $\equiv \text{Bidirectional}(I1, ..., \text{return\_sequences}, ..., \text{time-major})$, where $I1$ is an input tensor and $I1^T$ is the transpose of $I1$. This rule requires the `time_major` argument to be `False` (i.e., using `batch-major`) for the graph in Figure 1a and `True` for the equivalent graph in Figure 1b. We generalize the API function `Bidirectional` to all relevant API functions F, generalize input $I1$ to any input I, and generalize the configuration `return_sequences` so that the parameter `return_sequences` can be `True` or `False`, while only `True` was used in this concrete rule. The generalized equivalence rule is $(\text{F}(I^T, \text{ batch-major}))^T \equiv \text{F}(I, \text{ time-major})$. To make the generalized rules look cleaner, we generalize but omit in the rule notation all other parameters of the API functions including `return_sequences`, which is part of the configurations as explained in Section 2.

Based on our study of DL bugs and API documentation, we define 16 equivalence rules that can transform a graph into an equivalent graph. We group these rules into 6 categories (Table 1),

where I is a general input (often a tensor) and F denotes an API function. Function `implement(`F_A`, `F_B`)` is a function that uses function F_B to implement the functionality of function F_A. F_{2D} and F_{3D} are API functions that compute 2D and 3D operations (e.g., `tf.keras.layers.Conv2D` and `tf.keras.layers.Conv3D`) respectively. Function `dense` transfers input I to a dense tensor, while Function `sparse` transfers input I to a sparse tensor. Functions `normalize` and `denormalize` transfer image input I from float representation in range $[0, 1.0]$ to integer representation in range $[0, 255]$, and trasfer the function F's output back from integer to float. Function `cast` is a type-casting function that converts I to the expected data type, while $type_X$ and $type_Y$ are two different data types. Functions `decode` and `encode` are a pair of functions that decode an image file to a tensor or encode a tensor to an image file (e.g., `tf.io.decode_png` and `tf.io.encode_png`). Function `pad` denotes a padding function, while `unpad` is a function that reverses the padding procedure, (e.g., `tf.image.extract_glimpse`). Finally, M indicates a pretrained DL model, while `eval` is the model evaluation procedure.

Below, we first go through a detailed example of one rule, then describe the other rules.

3.1.1 A Detailed Example: Optimization Equivalence Rule. TensorFlow and PyTorch include several graph compilation optimizations that cause a function to be compiled as a callable graph. Compiling the program into callable graphs enables optimizations such as operation pruning or constant folding, which can significantly reduce execution time.

Optimization is known to cause many bugs [23, 41] in other domains (e.g., compiler optimization [20, 21, 35]). DL optimization (e.g., autograph transformation) is also complex and error-prone. For example, we found two non-crash bugs related to TensorFlow optimization by looking at GitHub issues (GitHub issue 47970[2]). Function `tf.math.floordiv` behaves differently with and without optimization, and so does `tf.linalg.eigh`.

Based on this bug report, we build two equivalent graphs (Figures 3) to reproduce the bug. Then we generalize the equivalence rule by abstracting the API `tf.math.floordiv`, the optimization `@tf.function`, and its input to build Rule 1 in Table 1. Rule 1 states
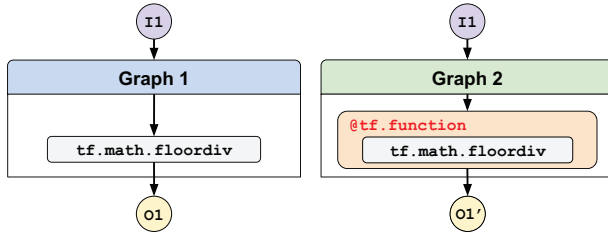
---

[2]https://github.com/tensorflow/tensorflow/issues/47970

Figure 3: Example of pair of concrete equivalent graphs generated from TensorFlow issue 47970.

Table 1: List of rules. F **is an API function,** I **is input, and** M **is a pretrained model.**

ID	Equivalence Rule
	Optimization
1	$F(I) \equiv \text{optimized}(F(I))$
	API Redundancy
2	$F(I, \text{padding=SAME}) \equiv F(\text{pad}(I,\text{SAME}))$
3	$\text{implement}(F_{2D}, F_{3D})(I) \equiv F_{2D}(I)$
4	$\text{implement}(\text{depthwise}, \text{conv2d})(I) \equiv \text{depthwise}(I)$
5	$\text{implement}(\text{separable}, \text{depthwise})(I) \equiv \text{separable}(I)$
6	$\text{implement}(\text{dilated}, \text{conv2d})(I) \equiv \text{dilated}(I)$
7	$\text{implement}(F, \text{document}_F)(I) \equiv F(I)$
	Data Structure Equivalence
8	$F(\text{dense}(I)) \equiv F(\text{sparse}(I))$
	Data Format Equivalence
9	$F(I) \equiv \text{denormalize}(F(\text{normalize}(I)))$
10	$(F(I^T, \text{batch-major}))^T \equiv F(I, \text{time-major})$
11	$F(I) \equiv F(\text{Dataset}(I))$
12	$F(\text{cast}(I, type_X)) \equiv F(\text{cast}(I, type_Y))$
	Inverse Equivalence
13	$\text{decode}(\text{encode}(I)) \equiv I$
14	$\text{unpad}(\text{pad}(I)) \equiv I$
	Model Evaluation Equivalence
15	$\text{eval}(M, I, \text{batch-size}=s1) \equiv \text{eval}(M, I, \text{batch-size}=s2)$
16	$\text{eval}(M, I) \equiv \text{save}(M), \text{eval}(\text{load}(M), I)$

that the computation of an arbitrary function F on input I is equivalent to the optimized version of this computation on the same input. Using this rule, EAGLE detects ten new bugs, including seven of them already confirmed or fixed by the developers. (Section 5.1).

3.1.2 Description of All Other Rules. We describe all other equivalence rules that we create category by category.

API Redundancy (Rules 2 to 7): The second category of rules concerns API redundancy, i.e., generating an equivalent graph using a different API. We identified several types of API redundancy.

Some APIs have built-in functionalities that can be executed externally. For example, many DL functions support built-in padding as an argument. SAME padding is a popular padding setting that produces an output of identical shape to the input when the stride is set to one. Therefore, using the built-in padding argument is equivalent to padding the input using SAME padding and then feeding the padded input to the function without using its padding option (**Rule 2**).

Many 2D functions (e.g., tf.keras.layers.Conv2D) can be implemented using the 3D version of the function by adding a dimension of length one to both the input and kernel, setting the stride to one for this dimension, and removing that dummy dimension from the output. These layers cover different API functions but should behave identically (**Rule 3**).

Rules 4 to 6 are about the reimplementation of advanced convolutions. There are many variants of advanced convolutions (e.g., dilated, depthwise, or separable). DL libraries provide built-in APIs for these advanced convolutions, but they can be reimplemented using other convolutions. For example, TensorFlow's DepthwiseConv2D function can be reimplemented using only Conv2D (**Rule 4**) by splitting the input and filters into X slices (X being the number of channels of the input) and computing the convolution for each slice of input and filter.

Finally, **Rule 7** leverages formulas found in API documentation to reimplement specific functions. For example, TensorFlow's tf.keras.layers.BatchNormalization's documentation states that "the layer (function) returns $gamma * (batch - mean(batch)) / sqrt(var(batch) + epsilon) + beta$." From this formula, two equivalent graphs can be created. The first one uses the API call to BatchNormalization, and the second one contains our reimplementation based on the documentation's formula. We generalize this example to obtain the following equivalence rule: when a formula is available in the API documentation, using the API should be equivalent to using the formula. While the formula will likely be implemented in some way in the API, the function likely contains additional control flow or conversion to handle exceptions or edge cases that might introduce inconsistencies.

Data Structure Equivalence (Rules 8): Many APIs take different types of data structures as input, and the functionality of such an API is identical regardless of the types of data structures used. For example, DL libraries often use tensors (multi-dimensional data structures) as input. These tensors can be represented as dense or sparse tensors. Sparse tensors are a tensor representation that is more efficient with mostly-empty tensors. DL libraries are expected to handle both representations either by having an API supporting both dense and sparse tensors or by providing an equivalent API specifically for sparse tensors. Therefore, given the same input, any function taking dense tensors should produce identical output to the same function (or its sparse version) taking a sparse tensor as input, with the only difference being computation time.

Data Format Equivalence (Rules 9 to 11): Data can be presented to DL APIs in different formats that can become equivalent with a few transformations.

For example, there are two principal ways to feed images to a DL network. In the first one, each pixel is represented as an integer (e.g., between 0 and 255 for the RGB file format). In the second one, each

Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan

pixel is represented as a floating-point number (e.g., between 0.0 and 1.0). One can normalize values between 0 and 255 to values between 0.0 and 1.0, and vice versa. Many different functions are supposed to support both types of representation without any casting. Given the same input in the two representations, either normalized or not, the outputs (one as is and one denormalized) of these functions should be equivalent (given a threshold related to floating-point imprecision). Thus, we create **Rule 9** to build equivalent graphs using these two types of image formats.

In addition to images, text is another common type of input to DL functions. Textual input can be fed to RNN either in time-major or batch-major format. Thus, we create the equivalence rule (**Rule 10**) described in the Introduction (Figures 1a and 1b). While these two Figures display two concrete equivalent graphs for a specific TensorFlow function (`tf.keras.layers.Bidirectional`), this rule applies to any API that supports time-major and batch-major inputs.

DL libraries provide a specific class called `Dataset` in PyTorch and TensorFlow to support complex input pipelines for model training and evaluation. Input can be applied to DL functions in two ways: (1) the input I can be passed to the DL function F directly, or (2) the input can be transformed to a `Dataset` object before being fed to F. Transforming the input to a `Dataset` has several advantages, including many built-in APIs that can be used to interact with the `Dataset` efficiently. We generate an equivalence rule (**Rule 11**) based on these two ways of applying an input to a DL function.

Finally, DL libraries accept different input data formats that are often equivalent for specific data ranges. We build **Rule 12** based on this observation. We cast the input to two different data types, $type_X$ and $type_Y$, and then feed them to an API function. The outputs are expected to be the same, if the input is within the intersection of $type_X$ and $type_Y$'s ranges, unless data overflows occur. For example, if a function accepts both `int8` and `int16` integers as input, any `int16` input that falls within the `int8` range ([-128,127]) should produce an equivalent output to its `int8` counterpart's computation output. One exception is when the `int8` computation overflows, e.g., an addition of two `int8` numbers may not overflow `int16` but overflow `int8`, in which case an exception would be thrown.

Inverse Equivalence (Rules 13 to 14): Many DL APIs have inverse functions. We develop two rules based on two extremely common DL preprocessing steps that can be inversed: encoding and padding.

Many types of input (e.g., image, sound, and text) have multiple encoding types (e.g., gif and png for images). Many of these encodings are built-in in DL libraries and should therefore be tested thoroughly. Any input encoded then decoded with the correct lossless encoding and decoding algorithms should be equivalent to the original input (**Rule 13**).

Padding is widely used to enlarge the size of input. We can unpad the padded input by extracting a window of the original size from the padded input. The extracted input should be equivalent to the initial input (**Rule 14**).

Model Evaluation Equivalence (Rules 15 and 16): In inference mode, evaluating the same trained model on the same test data

should result in the same output (e.g., the same label for an instance or the same accuracy) independently of the batch sizes (**Rule 15**).

A model should behave equivalently (in terms of accuracy, loss function, and weights) before and after being saved and loaded, independent of how it was saved and loaded (**Rule 16**). Bugs in the saving and loading code can cause inconsistencies, thus enabling EAGLE to detect such bugs.

3.2 Equivalent Graph Construction

The equivalence rules presented in Section 3.1 are general and applicable to many DL API functions. The next step (step 2 in Figure 2) is to concretize these rules into specific graphs by replacing abstract elements of the rules (e.g., F and I) with concrete APIs, input, and configurations.

For each rule, we identify a list of relevant APIs for the DL library under test by referencing its documentation. EAGLE then concretizes the rules for each applicable API. For example, EAGLE concretizes Rule 1 to a graph by replacing F with the TensorFlow API `tf.math.xdivy` and "optimized" with `tf.function` (TensorFlow's graph compilation optimization). It is relatively straightforward to extract applicable APIs in the target library by using heuristics and regular expression matching, and then manually verify them.

Some rules apply to many APIs: for example, Rule 1 of EAGLE generates equivalent graphs for 960 TensorFlow and 435 PyTorch APIs. Other rules such as Rule 10 are only applicable to certain API functions: for example, PyTorch's documentation lists three main RNN functions that can be tested with Rule 10 (`torch.nn.RNN`, `torch.nn.LSTM`, and `torch.nn.GRU`). While testing only three APIs might not seem general, these high-level APIs support multiple configurations that will test different underlying APIs. For example, under some configurations, the `torch.nn.GRU` function might also call the `Dropout` or `Bidirectional` functions. Obtaining applicable APIs is a one-time cost, and it is fast using heuristics.

3.3 Bug Detection

The final step (step 3 in Figure 2) is to generate input, e.g., to concretize I in Rule 1, and compare the output of the concretized graphs given the same input. We use existing work D2C [43] to generate input automatically. For example, EAGLE further concretizes the I of Rule 1 for API `tf.math.xdivy(x,y)` to `[-3.e+38+0.j, 2.e+37-2.e+38j]` (Figure 4). We then compare the output of each pair of concrete equivalent graphs, given the same input. To mitigate the impact of non-determinism of DL computation, we use the same random seed for the two equivalent graphs' executions and report all inconsistent output above a threshold.

With the concrete function and input, EAGLE detects a previously unknown bug in TensorFlow 2.5 and 2.6 that developers confirmed (Figure 4 in Section 5.2).

The main contribution of EAGLE is equivalence rules, which can be used together with any other test generation approach. Section 5.3 shows that 20 bugs that EAGLE detects cannot be detected by the chosen test generation technique (i.e., D2C) without equivalent graphs. Since our goal is to detect hard-to-detect non-crash bugs (due to the oracle challenge [3]) by detecting inconsistent behaviors (as opposed to, for example, crashes due to mishandling

of invalid input), we need a technique that is capable of generating valid input for DL API functions.

Instead of manually writing input constraints, which is commonly used in testing [10, 27], we leverage D2C that analyzes relevant API documentation to extract input constraints, and uses the constraints to guide the generation of input. For example, given the API document sentence for `tf.math.xdivy(x,y)`—"A Tensor. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, or `complex128`", it randomly generates a tensor whose elements are of type `half`, `float32`, `float64`, `complex64`, or `complex128`. D2C extracts four categories of constraints: *structure* such as list, tuple, and n-dimensional array (i.e., tensor), *type* such as `int`, `float`, `boolean`, and `String`, *shape* such as two-dimensional (2-D) array, and *valid value* such as parameter `padding` can only be one of "zeros", "border", and "reflection".

D2C uses *sequential pattern mining* [13, 15] to mine frequently occurring patterns (e.g., "Must be one of the following types") in API documents and transforms them into rules (e.g., "Must be one of the following types <*type* 1>, <*type* 2>") to extract input constraints automatically. The precision and recall is 94.4% and 92.4% for TensorFlow and 95.6% and 93.5% for PyTorch. We then manually verify the extracted constraints and add any missing ones.

For test generation, given an API function and its extracted constraints, the technique aims to generate valid input following the extracted constraints. Specifically, it chooses a *type* from the list of *types* in the constraints and creates a *shape* following the constraints. If the constraints do not specify a list of valid *types*, the test generation selects one from *types* supported by the library. Finally, the *structure* constraints are checked. For example, if the generated value is 1-dimensional and the constraints explicitly specify the *structure* (e.g., tuple or list), the input generator converts the generated value accordingly.

We finally execute all inputs and report inconsistencies between equivalent graphs.

4 EXPERIMENTAL SETUP

In total, we investigate 1,542 issues in TensorFlow and PyTorch. For TensorFlow, we focus on issues in TensorFlow 2.X only. For both PyTorch and TensorFlow, we use the GitHub search engine for closed issues labeled as "bug" with the keywords "fix." Then we manually check all the issues to filter out crash-related issues. Out of these 1,542 issues, 35 are relevant non-crash bugs, from which we create and generalize rules. Many GitHub issues are not relevant because (1) they are not bugs, e.g., user mistakes or feature requests, and (2) many issues describe crash bugs. In total, we extract 16 equivalence rules.

For rules 1-14, we use D2C to generate inputs. We generate up to 400 inputs per API. We use D2C to generate inputs for 963 TensorFlow APIs and 464 PyTorch APIs. For rule 15 and rule 16, we save 18 TensorFlow Keras pretrained models and 12 PyTorch pretrained models for testing. For the input, we extract 1,000 images from the ImageNet dataset and preprocess them according to the models.

After we generate inputs, we define a list of applicable APIs for each rule by referencing the API documents. EAGLE uses these rules to generate equivalent graphs for each applicable API and uses the inputs generated to compute the results.

We consult the inconsistency threshold formula that TensorFlow and PyTorch use in their test suite to determine whether the two outputs from two equivalent graphs are equivalent. For example, for the equivalent graphs G_1 and G_2 with respective outputs $O1$ and $O1'$ (given input $I1$), their results are equivalent if $abs(O1, O1') <= atol + rtol * abs(O1')$, with $atol=10^{-2}$ and $rtol=10^{-5}$.

We evaluate EAGLE on TensorFlow 2.1, 2.2, and 2.3 and PyTorch 1.6 and 1.9 since they were the latest versions available when we started this project. We only report a bug to developers if we can reproduce the bug on the latest version of TensorFlow and PyTorch (TensorFlow 2.6 and PyTorch 1.9) at the time of writing.

We obtain the total number of bugs by considering all inconsistencies for each rule and API pair as one bug. For example, in Rule 16, if five different models display inconsistencies to load with one API (e.g., `load_state_dict`), we only count it as one unique bug.

5 EVALUATION AND RESULTS

This section presents the results of our five Research Questions (RQs). RQ1 (Section 5.1) presents the number of bugs EAGLE detects. RQ2 (Section 5.2) describes some of the bugs for each category. RQ3 (Section 5.3) compares EAGLE to other DL testing approaches, and RQ4 (Section 5.4) explores how developers use equivalent graphs. Finally, RQ5 (Section 5.5) studies EAGLE's execution time.

5.1 RQ1: How many bugs does EAGLE detect?

We implement 16 rules to test the two most popular DL libraries, TensorFlow and PyTorch, resulting in 6,861 pairs of concrete equivalent graphs. We use previous work [43] to generate up to 400 sets of (input, configurations) per pair of equivalent graphs. A set of (input, configuration) consists of input to an API and its configuration (weights, etc.). For example, when testing the API `tf.keras.layers.Dense`, the input is a Tensor, and the configurations include weights, kernel initializer, and bias regularizer. For each set of input and configuration values, we compare the corresponding equivalent graphs.

Table 2 displays the number of bugs found in TensorFlow and PyTorch. Overall, EAGLE generates 6,861 pairs of equivalent graphs and detects 1,212 inconsistencies automatically. Multiple inconsistencies that are triggered by the same API function (with different inputs) are counted as one bug. As a result, these inconsistencies map to 25 bugs, including 13 previously unknown bugs (Table 2). Most (9) of these previously unknown bugs have been confirmed or fixed by TensorFlow or PyTorch developers. EAGLE also detects crashes for 42 APIs, among which we have only manually verified five since our focus is on non-crash bugs, which existing techniques have a hard time detecting.

Table 2 also shows the number of bugs found in each rule category. For example, Optimization is the category for which EAGLE finds the most number of bugs, with a total of ten bugs found. All those ten bugs are previously unknown bugs, seven of which have been confirmed or fixed by the developers. Section 5.2 describes examples of bugs found by EAGLE.

Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan

Table 2: Bugs found by each rule category.

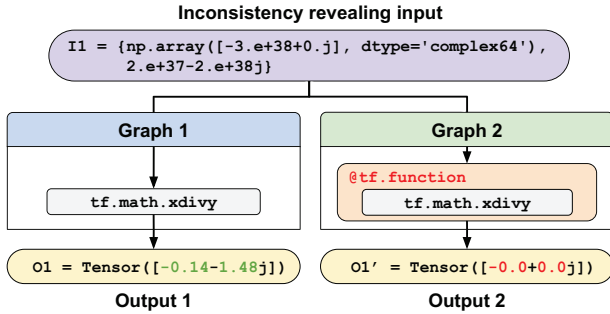| Category	TensorFlow	PyTorch	Sum
Optimization	10	0	10
API Redundancy	0	0	0
Data Structure Equivalence	3	4	7
Data Format Equivalence	1	3	4
Inverse Equivalence	2	0	2
Model Evaluation Equivalence	2	0	2
Total	**18**	**7**	**25**

Figure 4: Two equivalent graphs that detect a new inconsistency bug in TensorFlow, which has been confirmed by developers after we reported it.

Summary: EAGLE detects 25 bugs in the most widely-used DL libraries TensorFlow and PyTorch, including 13 previously unknown bugs, nine of which have already been confirmed or fixed after we report them.

5.2 RQ2: What bugs are detected by EAGLE?

We describe non-crash bug examples in each category of rules.

Optimization: EAGLE detects ten bugs that are revealed by inconsistencies between a standard graph and an optimized graph. All of these bugs are previously unknown bugs for which optimized TensorFlow API functions generate incorrect outputs. Figure 4 shows an example of a new bug in the `tf.math.xdivy` API detected by EAGLE that TensorFlow developers confirmed after we reported it. The annotation `@tf.function` on Graph 2 tells TensorFlow that the function below should be optimized. According to TensorFlow developers, this bug is caused by an overflow for `complex64` divisions in the optimization.

Data Structure Equivalence: With the rules of data structure equivalence, EAGLE detects three bugs in TensorFlow and four bugs in PyTorch. Figure 5 displays two equivalent graphs that EAGLE generated, which revealed a bug in PyTorch. API functions `torch.addmm` and `torch.sspaddmm` perform the same computation for dense and sparse tensors, respectively. Given three input tensors, $T1$, $T2$, and $T3$, these functions multiply $T2$ and $T3$, then add $T1$ to the result. The bug was deep in the C++ backend code of `torch.sspaddmm` in a low-level function (`indices.data_ptr`) that
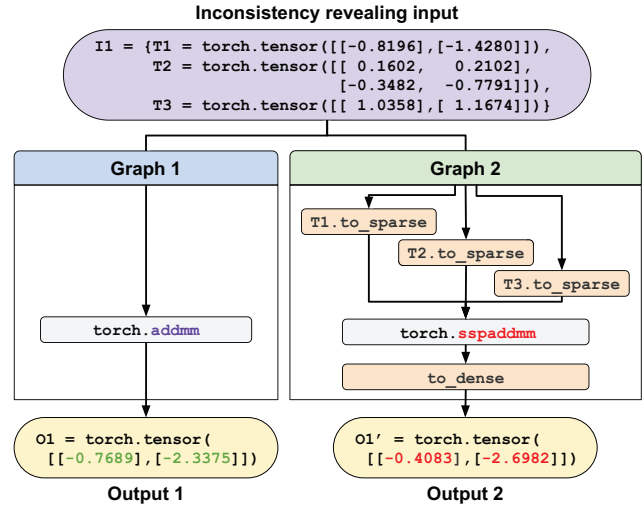

Figure 5: Pair of equivalent graphs that detects an inconsistency bug in PyTorch.

assumes row-contiguous storage of tensors, while `torch.sspaddmm` used another type of storage. The APIs under test (`torch.addmm` and `torch.sspaddmm`) do not have a direct counterpart in TensorFlow, so it would be very difficult to find this bug using cross-library differential testing techniques such as CRADLE or Audee.

Data Format Equivalence: EAGLE detects four bugs in this category, including the bug in Figure 1. The other three bugs are inconsistency bugs in three different PyTorch APIs. In `torch.fmod` and `torch.remainder`, there is a large inconsistency between equivalent `int64` and `float64` input while the `cosine_similarity` API has inconsistencies between `int8` and `int16`. These are bugs in the C++ low-level tensor library (ATen) used by PyTorch.

Inverse Equivalence: EAGLE detects two bugs in this category, including one new bug related to the `tf.io.decode_gif` API. Gif encoding is supposed to be lossless, but we found that in TensorFlow, for specific inputs, this encoding is not lossless, i.e., encoding and then decoding an input instance can result in significantly different outputs. The consequence of this bug is severe because image preprocessing is an essential part of many DL systems, and any bug in that preprocessing may modify the input to the DL models in an unexpected way that may lead to incorrect output, which would be hard to debug.

Model Evaluation Equivalence: EAGLE detects two bugs in this category. Those two bugs are inconsistency bugs in model configurations or metrics. For example, Rule 16 enables EAGLE to detect a bug in TensorFlow API `tf.keras.Sequential.from_config`. TensorFlow APIs `get_config` and `from_config` extract a model's configuration and build a model object from such a configuration respectively. Combined with `get_weights` and `set_weights`, they can achieve the functionality of saving and loading a model. Saving the model using `get_config` and `get_weights` and loading it using `from_config` and `set_weights` cause the model's configuration to be incorrect, which leads to the detected inconsistencies.

API Redundancy: EAGLE finds no bugs using the API redundancy rules. After investigating both TensorFlow and PyTorch libraries, we find a possible reason is that developers already implemented some rules from this category in their test suite after finding a bug in a previous version. For example, a concrete pair of equivalent graphs that EAGLE generates for **Rule 7** with the `BatchNormalization` API is included in the TensorFlow test suite. Similarly, we also found reimplementations of depthwise convolutions using `Conv2D` in the TensorFlow test suite (**Rule 4**). This demonstrates that developers are already using some equivalence rules to test their libraries as an afterthought of relevant bugs. A comprehensive set of rules and a technique that uses the rules to generate equivalent graphs and detects bugs would be beneficial for them to improve their testing system further.

False Positives: Out of all 26 inconsistent APIs detected by EA-GLE, we found one false positive (the other 25 are true bugs). This false positive is revealed by Rule 16 in PyTorch. When testing Rule 16 in PyTorch, we evaluate the pretrained model InceptionV3 before saving its internal states and after reloading them using `load_state_dict`. InceptionV3's input needs to be normalized and the normalization process is included along with the model architecture. When the pretrained weights are used, PyTorch not only loads the weights but also configures the model architecture by adding the input normalization process accordingly. However, the input normalization is not configured correctly after model saving and loading, which leads to the inconsistencies.

Generalizability of the Rules: All 16 rules apply to both Tensor-Flow and PyTorch, and Rule 8 finds bugs in both libraries. While a single rule can find bugs in both TensorFlow and PyTorch, it does not mean that these bugs can be found by cross-library differential testing techniques [14, 31], because when the rules are concretized to concrete graphs, the concrete APIs often only exist in one library. For example, Rule 8 finds bugs in both TensorFlow and PyTorch, but the API in which some of the bugs occur (`torch.sspaddm`) only exists in PyTorch.

In total, we generate 6,861 pairs of concrete equivalent graphs (429 pairs of graphs per rule on average) that are each tested on 400 sets of (input, configuration). The largest number of APIs covered by a unique rule is 963 and 464 for TensorFlow and PyTorch, respectively. Overall, the 25 bugs detected by EAGLE occur in very diverse APIs, from DL layers (`tf.keras.layers.Bidirectional`), low-level computation libraries (`torch.smm`), utility APIs (`tf.keras.Sequential.from_config`), optimization (`@tf.function`), or data preprocessing (`tf.image.decode_gif`).

> **Summary**: The 25 bugs detected by EAGLE in TensorFlow and PyTorch are in a very diverse set of DL APIs, including prepro-cessing, DL layers, low-level APIs, and utility functions, demon-strating the diversity and generality of our rules.

5.3 RQ3: Does EAGLE detect bugs not detected by other DL library testing techniques?

We compare EAGLE with two types of techniques that test DL libraries to better understand EAGLE's contribution. First, we com-pare EAGLE with a state-of-the-art fuzzing technique, D2C [43].

Second, we compare with two state-of-the-art differential testing techniques for DL libraries, CRADLE [31] and Audee [14].

Comparison with D2C [43] We ran D2C on the same PyTorch and TensorFlow versions on which we evaluated EAGLE. Although EAGLE uses D2C's input generation, only five of the bugs detected by EAGLE are also detected by D2C. D2C cannot detect any of the other bugs because it focuses on crash bugs, while the majority (20 out of 25) of the bugs found by EAGLE are non-crash bugs.

Comparison with CRADLE [31] and Audee [14] CRADLE and Audee are DL testing approaches that rely on Keras' high-level API to perform differential testing across libraries. Audee also has non-differential testing checkers, but since they do not detect in-consistencies, we focus on the differential testing aspect of Audee for this RQ).

Differential testing techniques such as CRADLE and Audee can-not detect bugs that EAGLE detects for the following reasons. Keras is a high-level library that allows users to build DL models in a back-end library-independent manner, i.e., one can seamlessly switch the backend DL library. Keras models can then be executed with-out reimplementation using different DL backends (TensorFlow, Theano, and CNTK). To do that, all backends must either implement the same functionalities, or Keras must implement missing features of backends.

With the explosion of DL in the last few years, DL libraries are growing fast, and many new types of DL functions are proposed that are not implemented in all libraries, making it extremely hard to maintain cross-backend execution in Keras (since functions unique to a DL library must be reimplemented for all libraries). In addition, new DL libraries have grown to be very popular (e.g., PyTorch and HuggingFace's Transformer) that are not supported by Keras, while libraries (Theano and CNTK) supported by Keras are no longer maintained. As a result, maintaining cross-backend support in Keras became unmanageable, and Keras dropped this feature in 2019, making it challenging to run differential testing techniques such as CRADLE or Audee. Reimplementing such a high-level library to allow differential testing would be extremely expensive and tedious. EAGLE addresses this challenge by requiring only one DL library to detect bugs.

It is possible to perform differential testing only on functionali-ties that are implemented identically in both libraries (e.g., Dense layer and Conv2D). However, doing so would miss many bugs, i.e., 15 of the 25 bugs (60%) that EAGLE detects. For example, the bug displayed in Figure 5 occurs in a PyTorch API that does not have a direct counterpart in TensorFlow.

In addition, even if we have a high-level library that supports cross-backend execution, CRADLE and Audee might still not find the remaining ten bugs that EAGLE detects because they focus on complete system testing (i.e., they take a full DL model as input and measure inconsistencies in accuracy). In contrast, EAGLE fo-cuses on single low-level API testing (unit testing) to find bugs buried deep in a DL library. For example, while all the inconsis-tencies reported by Audee concern incorrectly implemented DL layers (`ThresholdedReLU`, `DepthwiseConv2D`, `SeparableConv2D`, and padding implementation), EAGLE finds bugs in very low-level functionalities such as `ATen`, the low-level tensor library used by PyTorch (Section 5.2). CRADLE and Audee might miss these bugs

because they only produce inconsistencies at a system level for specific models and input.

The equivalence rules are orthogonal contributions, which can be combined with CRADLE or Audee to help them generate more DL models to test more DL library code. For example, CRADLE and Audee may use our Rule 1 to test optimization code cross libraries if a high-level library such as Keras is revamped.

Summary: The majority (20) of the 25 bugs detected by EAGLE are non-crash bugs, whose relevant APIs have little cross-library redundancy. Thus, it would be difficult for existing testing approaches to detect these bugs.

5.4 RQ4: Do DL library developers use equivalent graphs?

In this RQ, we investigate if our rules are new by studying if and how developers have been using equivalent graphs to test DL libraries. We manually examine TensorFlow and PyTorch's test suites and check if any test cases implement (or partially implement) our rules.

Most (15 of the 16) rules are not implemented or not fully implemented in PyTorch test cases: 13 rules are not implemented at all, while two rules (1 and 16) are implemented only for a few APIs. Only Rule 8 is implemented for all the APIs tested by EAGLE.

The majority (13 out of 16 rules) are not implemented or not fully implemented in TensorFlow test cases: nine rules are not implemented at all, four rules (1, 8, 15, and 16) are implemented for only a few APIs, and only three rules (4, 6, and 7) are implemented for all the APIs tested by EAGLE for that rule. Such test cases were created likely as an afterthought after a bug was found. For example, after finding a bug in `torch.sspaddmm` from GitHub issue 45113[3], developers implemented a test case to test `torch.sspaddmm` and its dense version `torch.addmm` in PyTorch 1.7.

The fact that developers use equivalent graphs to make sure a bug is fixed shows that such graphs are useful to test DL libraries. However, equivalent graphs have not been implemented proactively to create test cases (i.e., to find bugs). EAGLE offers a more complete list of equivalence rules to generate equivalent graphs that developers have not manually implemented and can therefore improve the reliability of DL libraries.

Summary: Most (13 out of 16) rules are not implemented in DL libraries' test suites. The few test cases that implement equivalent graphs were only implemented as an afterthought after a bug has been reported. This indicates that EAGLE complements developers' test cases and can detect bugs that would be hard to find manually.

5.5 RQ5: What is the run time of EAGLE?

Table 3 shows EAGLE's execution time. On average, it takes 33 minutes to execute a pair of equivalent graphs with a set of 400 (input, configuration) in TensorFlow and 26 minutes in PyTorch. In total, EAGLE executes 6,861 pairs of equivalent graphs. It is easy to execute graphs in parallel. For example, on our Xeon Gold 5120R CPUs (56 cores in total) and 512 GB of memory server, we execute 24 graphs at a time.

---

[3]https://github.com/pytorch/pytorch/issues/45113

Table 3: Execution Time of EAGLE

	TensorFlow	PyTorch
# of pairs of concrete graphs	5,817	1,044
# of (input, config) per graph	400	400
Time per pair (minutes)	33	26

6 THREATS TO VALIDITY

EAGLE does not find all bugs: Since we focus on detecting inconsistencies between equivalent graphs, EAGLE might miss bugs that do not cause inconsistent outputs. For example, if a rule generates a pair of equivalent graphs that use two redundant APIs that contain the same bug, EAGLE will not detect the inconsistency. However, EAGLE is effective in detecting 25 bugs in TensorFlow and PyTorch automatically.

Manual rule construction: The rules to generate equivalent graphs have been manually designed. As a result, they might not be fully representative of real bugs in DL systems. To mitigate this issue, we look at existing bug reports in two popular DL libraries (TensorFlow and PyTorch) when designing our rules. Our results show that the rules designed for EAGLE find 13 previously unknown bugs, showing that they can be used to detect new real-world bugs.

Generability to different DL libraries: Our approach might not be generalizable to other DL libraries. To mitigate this threat, we evaluate EAGLE on the two most popular DL libraries, TensorFlow and PyTorch. EAGLE finds bugs in both libraries. In the future, we could further extend EAGLE to test different libraries (e.g., DeepLearning4J) to show EAGLE's generalizability.

Potential bugs in our implementation: Our implementations might be buggy. If that is the case we will either (1) incorrectly detect inconsistencies or (2) not detect the inconsistency. We mitigate (1) by manually looking at the inconsistencies we detect before considering them as bugs. None of the inconsistencies EAGLE finds are the result of a bug in our code. In addition, developers confirmed nine of the 13 new bugs EAGLE detects. For (2), our approach might not detect some bugs because of issues in our implementation. However, this can only hurt our results and therefore does not impact the validity of our findings. If bugs in our code cause us to miss inconsistencies, our technique might perform even better once we fix them.

Nondeterminism: Not all inconsistencies are bugs because DL APIs can be nondeterministic [32]. We address nondeterminism by fixing the random seed to make API testing reproducible. We also use a threshold used by popular DL libraries to take into consideration floating-point precision inconsistencies. Overall, all but one inconsistencies that EAGLE detects are the result of true bugs.

7 RELATED WORK

DL library testing suffers from the oracle problem. Specifically, DL API functionalities are very complex, and it is often hard to know or even approximate the expected output manually. Previous work [28, 48] shows that such oracle approximations are often used in DL libraries but are error-prone, resulting in flaky tests or requiring a manual update from the developers.

Fuzzing and differential testing can be used to mitigate the oracle problem. Fuzzing often only detects crashes, while differential testing generally requires two different libraries that implement the same functionality, which is difficult to achieve and error-prone. Our work is different since we leverage within library equivalences such as API redundancy or optimization to build equivalent graphs to detect non-crash bugs. Since the graphs EAGLE uses are equivalent, they both have the same expected output, addressing the oracle problem. To the best of our knowledge, we are the first to propose equivalent graphs and to use them to find 25 bugs in DL libraries.

Differential testing of DL libraries: Previous work [6, 14, 28, 31, 34, 39, 42] uses differential testing to find inconsistencies between DL libraries. Such inconsistencies are often the result of a bug in DL libraries. For example, CRADLE [31] finds bugs in Keras by running the same model with different DL backends (TensorFlow, Theano, and CNTK).

These approaches require either (1) a high-level library that supports several DL backends (e.g., Keras), (2) a good model converter (e.g., MMdnn), or (3) heavy engineering to reimplement the same DL computation in different DL libraries. Unfortunately, while Keras initially supported several backends and was used in previous studies [14, 28, 31, 42], Keras now only supports TensorFlow. MMdnn[24] or ONNX[2] are frameworks that allow transferring models across DL libraries, but MMdnn only supports a few popular layers (e.g., RNN layers are not supported), and PyTorch, one of the most popular DL libraries, cannot execute ONNX models. Therefore, the only solution for thorough differential testing across DL libraries is to reimplement the DL computation in different frameworks, which is time-consuming and error-prone. For example, previous work [34] only reimplemented two ML algorithms (K-Nearest Neighbours and Naive Bayes) when using differential testing on Weka, Rapid Miner, and KNIME.

In contrast, EAGLE uses equivalent graphs to find bugs in DL APIs, which is not limited by third-party libraries (converter or high-level API support). For example, EAGLE detects a bug in biRNN layers of TensorFlow, which would not have been found by differential testing using MMdnn or Keras since MMdnn does not support biRNN layers and Keras does not support multiple backends anymore.

Fuzzing DL libraries: Fuzzing is another popular approach to test DL networks. Classic fuzzing techniques [11, 25, 30] can be used to find some crash bugs, but more advanced fuzzing techniques targeting DL systems have been proposed [29, 43, 45, 47]. We use the approach developed by Xie et al. [43] to generate valid inputs for our approach; however, these fuzzing approaches still suffer from the oracle problem and can only find crash bugs (see Section 5.3). For example, previous work [43] could only find five of the 25 bugs detected by EAGLE; hence our approach complements existing fuzzing techniques.

Some of ProbFuzz's [6] checkers use differential testing and can detect non-crash bugs in probabilistic systems. They use across library differential testing or several implementations of the same API in different languages (e.g., Py-Stan and R-Stan). Similar to other differential testing techniques, ProbFuzz requires multiple libraries implementing the same functionality and has some scope limitations (e.g., ProbFuzz does not support loops) that make it difficult to apply to DL libraries. EAGLE tests DL APIs generally and aims at finding general bugs that ProbFuzz does not cover.

Other work testing DL libraries: Static analysis has been used to detect specific types of bugs (e.g., shape-related bugs) in DL systems [19]. EAGLE finds very diverse bugs in DL systems (Section 5.2) that are hard to find without equivalent graphs. Metamorphic testing has also been used to test and validate ML classifiers [5, 7, 44]. These approaches have only found injected bugs in ML systems, and previous work shows that injected bugs often only have a weak correlation with real bugs [12].

Equivalent graph generation: TASO [16] automatically generates graph substitutions to optimize a given deep neural network computation graph. It generates equivalent graph substitutions based on a given architecture and finds the one with the least inference time among all the substitutions. While TASO generates equivalent graphs, it does not use them to find bugs; instead, it uses equivalent graphs to optimize DL computations. Most of the TASO equivalence rules are mathematical equivalence rules such as for any tensors A, B, and C of concrete shape, $(A \otimes B) \otimes C = A \otimes (B \otimes C)$, where $\otimes$ denotes matrix multiplication. We implemented eight of the TASO rules and none detected any bugs. We focus on building rules that are inspired by real bugs and API documentation. All of the rules that we design for EAGLE are new, different from the ones in TASO.

Differential testing for compilers: Differential testing has been used for testing compilers [20, 21, 35]. Instead of equivalent graphs, these work generate equivalent programs modulo input (EMI). The key in EMI is to create a collection of correct programs that have the same output given the same input (but might have different outputs for other inputs. Our work is different since program compilation is a different problem than DL graph execution which presents its own challenges.

8   CONCLUSION

We propose and evaluate EAGLE, a new differential testing approach that uses equivalent graphs to test a single DL library. We design 16 new equivalence rules that can generate pairs of equivalent graphs. We evaluate EAGLE on the two most popular DL libraries, TensorFlow and PyTorch, and found 25 bugs, 13 of them are previously unknown bugs, and nine have already been confirmed or fixed by developers. In the future, the rules we describe could be combined to detect bugs in more complex API interactions within DL libraries.

ACKNOWLEDGEMENT

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[3] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[4] François Chollet et al. 2015. Keras. https://keras.io.

[5] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a Deep Learning Framework by Metamorphic Testing. In *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET '17)*. IEEE Press, 28–34.

[6] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. 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*. 574–586.

[7] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/3213846.3213858

[8] Amir Efrati. 2018. Uber Finds Deadly Accident Likely Caused by Software Set to Ignore Objects on Road. *The information* (2018).

[9] Simos Gerasimou, Hasan Ferit-Eniser, Alper Sen, and Alper Çakan. 2020. Importance-Driven Deep Learning System Testing. In *ICSE*.

[10] P. Godefroid, A. Kiezun, and M. Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*. 206–215.

[11] Google. 2021. *OSS-Fuzz*. https://github.com/google/oss-fuzz

[12] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How close are they to real faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 189–200.

[13] Karam Gouda, Mosab Hassaan, and Mohammed J Zaki. 2010. Prism: An effective approach for frequent sequence mining via prime-block encoding. *J. Comput. System Sci.* 76, 1 (2010), 88–102.

[14] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 486–498.

[15] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. Citeseer, 215–224.

[16] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[17] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.

[18] Keras. 2019. Keras 2.3.0: This is also the last major release of multi-backend Keras. https://github.com/keras-team/keras/releases/tag/2.3.0.

[19] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static analysis of shape in TensorFlow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[20] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[21] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[22] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting Operational DNN Testing Efficiency through Conditioning. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[23] J. Liang, Y. Chen, M. Wang, Y. Jiang, Z. Yang, C. Sun, X. Jiao, and J. Sun. 2019. Engineering a Better Fuzzer with Synergically Integrated Optimizations. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 82–92. https://doi.org/10.1109/ISSRE.2019.00018

[24] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1320–1330.

[25] LLVM. 2021. libFuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html

[26] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *ASE*.

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

[28] Mahdi Nejadgholi and Jinqiu Yang. 2019. A study of oracle approximations in testing deep learning libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 785–796.

[29] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. PMLR, 4901–4911.

[30] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[31] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[32] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: an analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 771–783.

[33] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2135–2135.

[34] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-implementation testing of supervised learning software. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.

[35] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

[36] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *ASE*.

[37] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering*.

[38] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2020. Testing DNN Image Classifier for Confusion & Bias Errors. In *ICSE*.

[39] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering discrepancies in numerical libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 488–501.

[40] Huiyan Wang, Jingweiu Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. DISSECTOR: Input Validation for Deep Learning Applications by Crossing-layer Dissection. In *ICSE*.

[41] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 147–159.

[42] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. 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*. 788–799.

[43] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael Godfrey. 2021. Leveraging Documentation to Test Deep Learning Library Functions. (2021). arXiv:cs.SE/2109.01002

[44] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558.

[45] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM*

SIGSOFT International Symposium on Software Testing and Analysis. 146–157.

[46] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*.

[47] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–412.

[48] Wujie Zheng, Wenyu Wang, Dian Liu, Changrong Zhang, Qinsong Zeng, Yuetang Deng, Wei Yang, Pinjia He, and Tao Xie. 2019. Testing untestable neural machine translation: An industrial case. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 314–315.

[49] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Lingming Zhang, Bei Yu, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *ICSE*.