




# D<sup>3</sup>: Differential Testing of Distributed Deep Learning With Model Generation

Jiannan Wang , Hung Viet Pham, Qi Li , Lin Tan , *Senior Member, IEEE*, Yu Guo, Adnan Aziz, and Erik Meijer

**Abstract**—Deep Learning (DL) techniques have been widely deployed in many application domains. The growth of DL models' size and complexity demands distributed training of DL models. Since DL training is complex, software implementing distributed DL training is error-prone. Thus, it is crucial to test distributed deep learning software to improve its reliability and quality. To address this issue, we propose a *differential testing technique*—D<sup>3</sup>, which leverages a *distributed equivalence rule* that we create to test distributed *deep learning software*. The rationale is that the same model trained with the same model input under different distributed settings should produce equivalent prediction output within certain thresholds. The different output indicates potential bugs in the distributed deep learning software. D<sup>3</sup> automatically generates a diverse set of distributed settings, DL models, and model input to test distributed deep learning software. Our evaluation on two of the most popular DL libraries, i.e., PyTorch and TensorFlow, shows that D<sup>3</sup> detects 21 bugs, including 12 previously unknown bugs.

**Index Terms**—Software testing, distributed deep learning, differential testing, model generation.

## I. INTRODUCTION

DEEP learning systems are pervasive. They have been widely deployed in many domains including recommendation systems [1], [2], self-driving cars [3], machine translation [4], [5], and language representation [6], [7].

Given the complexity and large sizes of DL models [2], [6], [7], [8], *distributed* DL training is required for many real-world DL systems. For example, the Generative Pre-trained Transformer 3 (GPT-3) model [7], which is an autoregression DL model that generates human-like text, has 175 billion parameters and takes up 350GB of space. An implementation

Received 20 November 2023; revised 30 July 2024; accepted 29 August 2024. Date of publication 16 September 2024; date of current version 10 January 2025. This research was supported in part by NSF 1901242, NSF 2006688, and a Facebook Research Award. Recommended for acceptance by P. Pelliccione. (*Corresponding author: Lin Tan.*)

Jiannan Wang, Qi Li, and Lin Tan are with the Computer Science Department, Purdue University, West Lafayette, IN 47907 USA (e-mail: wang4524@purdue.edu; li4246@purdue.edu; lintan@purdue.edu).

Hung Viet Pham is with the Electrical Engineering and Computer Science Department, York University, North York, ON M3J 1P3, Canada (e-mail: hvpham@yorku.ca).

Yu Guo, Adnan Aziz, and Erik Meijer are with Meta Inc., Menlo Park, CA 94025 USA (e-mail: yuguo@fb.com; adnanaziz@fb.com; erik.meijer@meta.com).

Digital Object Identifier 10.1109/TSE.2024.3461657

of a deep learning recommendation model (DLRM) [2] contains about 23 billion parameters, and the size of the model is 91.1GB. Training such a large DL model is time and space expensive. First, it typically takes weeks or months to train such models. For example, the time required to train the GPT-3 model with 175 billion parameters is 34 days on 1,024 GPUs [8]. In addition, such a model is too large to fit in a single GPU. As a result, it is mandatory to train such large models on multiple processors (e.g., GPUs or CPUs). This method is called distributed training. In distributed training, the training task is split and sharded among multiple processors, and each processor only handles part of the workload. By doing so, it not only makes it possible to train models that are too large to fit in a single processor but also speeds up the training process.

Correctly and efficiently splitting, sharding, and aggregating models and data at a large scale is difficult, contributing to the complexity of distributed DL training and inference [9], [10], [11]. Thus, software implementing distributed DL training and inference is error-prone [12], [13], [14], [15], [16], [17], [18], [19], [20]. In consequence, it is crucial to test distributed DL software to improve its reliability and quality.

### A. Challenges and Our Approach

Our goal is to detect bugs in distributed DL software, i.e., detecting implementation bugs in the code that defines, trains, and evaluates distributed DL models, including the backend code in the DL libraries. This goal is different from existing papers [21], [22], [23], [24] that aim to find erroneous behaviors in the trained models instead of the code that builds and trains models. Previous papers have shown that DL software bugs lead to incorrect output and failures despite correct model output [25], [26].

There are two main challenges in testing distributed DL software. The first challenge is that it is particularly difficult to know the expected output of DL programs, due to their complexity and large sizes [25], [26]. Existing techniques address this challenge by cross-checking different libraries or execution graphs to detect inconsistency bugs [25], [26], [27], [28], [29], [30]. None of the existing techniques is specialized in detecting bugs in distributed DL code. The second challenge is that exposing bugs in distributed DL libraries requires a large, diverse set of DL models to exercise distributed DL code.

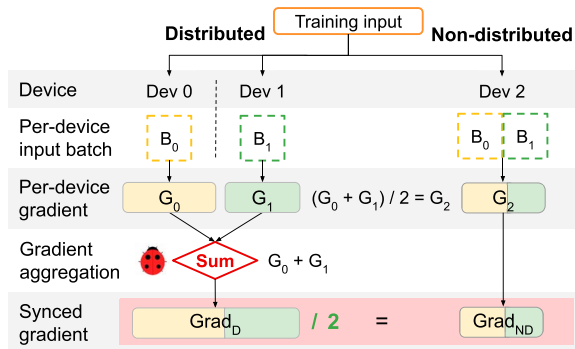


Fig. 1. D<sup>3</sup> detects a real-world bug, revealed by an inconsistency between a distributed setting and a non-distributed setting. The buggy code uses *sum* instead of *average* to aggregate gradients. The bar lengths of  $G_0$ ,  $G_1$ ,  $G_2$ ,  $Grad_D$ , and  $Grad_{ND}$  represent the magnitude of the gradient values. The bug in the gradient aggregation leads to  $Grad_D$  being twice as large as  $Grad_{ND}$ , which is fixed by applying the average (“/2” in green) to the synchronized gradient.

**Distributed equivalence rules:** To address the first challenge, we build a *differential testing* technique—D<sup>3</sup>, which leverages a *distributed equivalence rule* that we create to test distributed deep learning software. An equivalence rule defines specific conditions in deep learning libraries where different executions lead to equivalent output. We define our new distributed equivalence rule as that *the distributed training and inference should produce output that is equivalent to that of the non-distributed training and inference counterparts*. This rule also implies that the output of distributed training and inference with two distributed settings should also be equivalent.

Suppose that we start with the same DLRM model structure and train it with the same training instances. When trained on two GPUs (referred to as a *world size* of two), the resulting DLRM model is  $M_1$ , and when trained on four GPUs (*world size* of four), the resulting DLRM model is  $M_2$ . Here world size is the number of processors in the distributed cluster, e.g., the number of GPUs when training using GPUs. To make the large models fit in GPU memories,  $M_1$  is stored and trained on two GPUs, while  $M_2$  is stored and trained on four GPUs. Our distributed equivalence rule states that given the same input instance, model  $M_1$  and model  $M_2$  should produce equivalent output, e.g., two classification vectors with differences within small thresholds. If there is a bug in the DL training and inference code, e.g., in the PyTorch [31] or TensorFlow [32] libraries or the user code setting up these models, the output may be different, indicating software bugs. Our distributed equivalence rule enables us to detect such bugs in distributed DL software without knowing the expected output of a given input instance.

There are more distributed parameters than just the world size. For example, one can shard a model to multiple GPUs using different schemes for model parallelism, e.g., *column-wise sharding* splits an embedding table by its embedding dimension, and *row-wise sharding* splits the embedding table by its first dimension. Our distributed equivalence rule states that a model trained with column-wise sharding should produce output equivalent to a model trained with row-wise sharding.

**Distributed parameters:** We identify a diverse set of *distributed parameters*, i.e., world size, sharding type, device,

weight quantization, activation quantization, and sharder type. D<sup>3</sup> then uses these distributed parameters to generate a full set of distributed settings, to cross-check the equivalence of model training and inference to detect bugs in distributed DL software. Here a *distributed setting* consists of one concrete value for each and all distributed parameters, e.g., {world size: 2, sharding type: column wise, device: gpu, weight quantization: int8, activation quantization: float16, sharder type: EmbeddingBagSharder} is one distributed setting. Section III-C describes the candidate values for distributed parameters.

**Distributed model architectures and model input:** To address the second challenge, we design and implement a DL model generation technique that is specialized in producing distributed DL models and input to these models automatically for testing distributed DL software. Our approach generates a diverse set of DLRM models, chain structure models, and cell-based structure models.

### B. A Motivating Example

Our tool D<sup>3</sup> detects a severe bug automatically in the production DL software using PyTorch that affects a multi-national company. PyTorch’s large-scale distributed recommendation system TorchRec [33] uses different gradient aggregation strategies, e.g., the sum or the average, when aggregating gradients from the processors in a distributed cluster. By default, the company’s production DL models mistakenly use the *sum* instead of the *average* of gradients from the processors in the distributed training. Given the large sizes of these models, they are trained on many processors, e.g., 128 GPUs. After being put into production, each model is trained continuously on a smaller number of processors, e.g., 64 or 32 GPUs, since the volume of input data for incremental updating is lower. This leads to inconsistent gradient values, because the sum of gradients from 64 GPUs is smaller than the sum of gradients from 128 GPUs. This kind of bug causes regression in model accuracy, i.e., model accuracy is lower than before, in the training process and leads to revenue losses. Loading a model from an  $N$  node setting to an  $M$  node setting, where  $M < N$ , is tricky, and this bug is an example of that. Section V-B *Bug 1* describes this bug in detail.

Fig. 1 shows that D<sup>3</sup> detects this bug by generating a model and its model input and comparing the training on different numbers of GPUs (we use two GPUs versus one GPU to illustrate it without losing generality). Dev 0 and Dev 1 denote the two GPUs, while Dev 2 denotes the single GPU. The model’s training input is split into two batches (denoted by  $B_0$  and  $B_1$  in Fig. 1) when trained on two GPUs. When the same model is trained on one GPU, the same training input is now processed entirely on Dev 2. We use  $G_i$  to represent the per-device gradients on Dev  $i$ .

During the non-distributed training, TorchRec computes gradient  $G_2$ , which is a form of average metric (detail in Section V-B *Bug 1*) of all input instances in  $B_0$  and  $B_1$ . For the distributed training, TorchRec computes  $G_0$ , which is a form of the average metric of input instances in  $B_0$ , while  $G_1$  is for  $B_1$  instances. In the next step, TorchRec should compute the synchronized gradient  $Grad_D$  by calculating the *average* of  $G_0$  and  $G_1$ , which should be equal to  $G_2$  (small floating-point imprecisions allowed). But it computes the *sum* of  $G_0$  and  $G_1$

by mistake, which is twice as large as that from non-distributed training on a single device (Dev 2) ( $Grad_{ND}$ ). In summary,  $Grad_D$  equals  $G_0 + G_1$ , while  $Grad_{ND}$  is  $(G_0 + G_1)/2$ , although they are expected to be mathematically the same. The inconsistent gradient values reveal this bug.

$D^3$  automatically generates *test scenarios* to trigger the bug. A test scenario is a tuple of  $(S_k, M_i, I_{ij})$ , where  $S_k$  is a distributed setting,  $M_i$  is a model, and  $I_{ij}$  is the  $j$ th input instance to the model  $M_i$ .  $D^3$  then compares the inference output of the two resulting models from distributed training under every pair of distributed settings, given the same input instance. Among others,  $D^3$  detects an output inconsistency when comparing one distributed setting and one non-distributed setting with two processors and one processor respectively which leads to the detection of this bug.

Detecting this bug is hard for several reasons. Firstly, triggering this bug requires a sharding scheme for *model* parallelism, e.g., table-wise or row-wise. This bug does not happen with the *data* parallelism sharding scheme, which splits data instead of the model to multiple processors. This is because the data parallelism sharding scheme by default uses the average instead of the sum of the gradients from processors, resulting in consistent gradients. Secondly, triggering this bug also requires that a model contains certain types of layers such as TorchRec's `EmbeddingBagCollection`, which are lookup tables that convert a layer's input to a fixed length of vectors. Finally, the gradient differences are small and do not cause big accuracy drops without long-running training. Thus, without comparing the output of two distributed settings, this bug with severe consequences remained unnoticed.  $D^3$  automatically generates test scenarios with the specific model and sharding scheme to trigger this bug and compares the model output of different test scenarios to detect this bug.

### C. Contributions

In this paper, we make the following contributions: 1) We build the first *differential testing technique*  $D^3$  that is specially designed for testing *distributed* DL software. 2) We define a new *distributed equivalence rule* to address the oracle challenge of testing distributed DL software. We identify a diverse set of *distributed parameters* and we use and combine them to automatically generate distributed settings for testing distributed DL software. 3) We design a *model generation* method that generates models specifically for distributed DL software, and 4) Our *evaluation* of  $D^3$  on two widely used distributed DL systems PyTorch and TensorFlow shows that  $D^3$  detects 21 bugs, 12 of which are previously unknown bugs.

**Availability:**  $D^3$  code is available in the GitHub repo (<https://github.com/lt-asset/D3>).

## II. BACKGROUND

### A. Deep Learning Model

A DL model structure is typically represented as a directed acyclic graph (DAG) [34], which consists of nodes or layers that are connected to perform a specific task (e.g., regression or classification). Each layer applies a mathematical function (e.g., linear, embedding, convolution, etc.) to the input data with

specific weights. Specifically, the same type of layers can be adapted multiple times in a DL model structure. The operations performed on those layers are generally different because these layers have different parameters.

A DL model consists of its model structure and weights. To obtain the correct weights for each layer in a DL model, the model needs to be trained on a training dataset [35]. This is called the training phase. Once the training phase is finished, the weights (or parameters) of each layer are fixed and do not change. Then the model can be used in the inference phase. For evaluation, the trained model needs to be evaluated in the inference phase on a test dataset [36], [37]. The test dataset contains data different from the training dataset so that we can assess the model's performance and generalization.

DL models usually take tensors [31], [32], which are high-dimensional data structures, as input. The shape of a tensor is the length (number of elements) of each of the dimensions of the tensor.

### B. Distributed Training

Increasing data size and model complexity can generally lead to better model performance. However, the training process is very computation-intensive and thus time-consuming given the complexity and large sizes of DL models. Distributed training [38] is introduced to reduce the training time where the power of multiple processors is exploited to accelerate the training process, which is known as parallelism [39], [40].

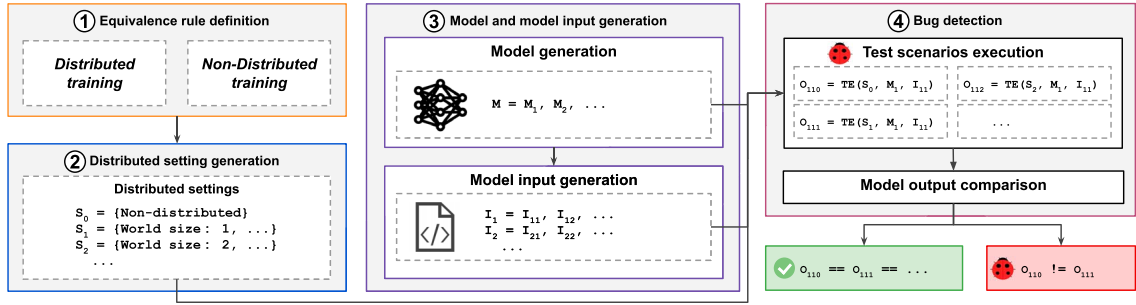
Specifically, there are two main types of parallelism schemes—Distributed Data Parallelism (DDP) [39] and Distributed Model Parallelism (DMP) [41], [42]. When utilizing DDP, the parameters are replicated on each distributed device, and the dataset is split into  $N$  parts, where  $N$  is the number of distributed devices, e.g., GPUs. During training, each device calculates its gradients using local parameters and data. In the end, the gradients on each device will be aggregated as the final gradients.

For DMP, each part of the model is placed on different devices. The DMP setting is widely used for very large deep-learning models when the model cannot fit into a single GPU's memory. For example, deep learning models in the recommendation system usually have very large embedding layers to handle the high-dimensional input. Sometimes the model is too large for a single GPU. With DMP, the huge embedding layers are split and then distributed to multiple GPU devices, which makes training and deployment possible.

It is non-trivial to split the model or dataset into parts and assign each part to different computing processors. Therefore, it is of vital importance to ensure the correctness of distributed training and inference of deep learning models.

### C. Deep Learning Recommendation Model (DLRM)

To test distributed DL libraries, we focus on a specific type of model: the Deep Learning Recommendation Model (DLRM) [2]. DLRM is tailored for recommendation systems and aligns well with the requirements of distributed training and inference. In recommendation systems, models must handle categorical data, such as user demographics (e.g., gender, age group, occupation), and dense features, such as item prices and user

Fig. 2. D<sup>3</sup> overview.

ratings. Consequently, a DLRM contains multiple embedding bags that map categorical data to dense representations in an abstract space and a multilayer perceptron (MLP) that processes dense features. These features are then combined and further processed through another MLP to compute event probabilities. Given that recommendation systems rely on vast amounts of data and require sophisticated models, DLRM, which was specifically designed for this domain, is an ideal candidate for distributed training and inference, encompassing both model parallelism and data parallelism.

#### D. Differential Testing

Finding bugs, especially non-crash bugs in deep learning libraries is nontrivial because it is hard to know the expected output given the increasing complexity of deep learning algorithms. We cannot use the ground truth as the expected output because the deep learning model is not 100% correct. When the model makes mistakes on certain input, the expected output of the model is not the ground truth. In addition, the nondeterminism in the DL computation makes the same model trained on the same input have different output [43]. For example, when using multi-process data loading, it is hard to load data in the same order [44], which makes training with the same model and input have different prediction results. Such nondeterminism adds difficulty to differential testing.

Recent studies address this challenge using differential testing [25], [26], [27], [28], [45]. It uses at least two implementations of the same functionality to produce equivalent output given the same input. Inconsistencies between output indicate potential bugs. This method automates testing processes, saving time and effort.

The key to applying differential testing to test deep learning libraries is to find the equivalent components expected to produce the same output given the same input. In this project, we define an equivalence rule and generate equivalent distributed settings. When the same model is distributed under those settings, we expect equivalent output when feeding the same input to the model. We use the equivalence rule as the oracle to detect inconsistency bugs in distributed deep learning libraries.

### III. APPROACH

In this section, we describe how D<sup>3</sup> detects bugs in distributed DL libraries using the distributed equivalence rule.

#### A. Overview of D<sup>3</sup>

D<sup>3</sup> automatically generates *test scenarios*, i.e.,  $(S_k, M_i, I_{ij})$ . Note that we need different model input for different models because the input layers of different models have different shapes. D<sup>3</sup> then compares the inference output of the two resulting models from distributed training under every pair of two distributed settings, given the same input instance, to test distributed deep-learning software.

Fig. 2 presents the overview of D<sup>3</sup>, which automatically generates test scenarios for the differential testing of *distributed* deep learning software. D<sup>3</sup> consists of four steps. First, we create a new equivalence rule, which states that under certain distributed settings, the same model trained on the same model input should be equivalent, i.e., produces equivalent output when feeding with the same input (①). In the second step (②), we collect parameters for distributed settings and their candidate values. D<sup>3</sup> generates distributed settings by selecting one candidate value for each parameter. Those generated distributed settings are later used to train models. D<sup>3</sup> generates distributed settings for all possible combinations of the distributed parameters' candidate values that we collect.  $S_0$  represents the non-distributed settings (a special case of distributed settings), while  $S_k$  denotes all other distributed settings, where  $k > 0$ . For example,  $S_1$  represents the distributed setting that the world size equals one, the sharding type is table-wise sharding, the device is on CPU, weight quantization is float32, activation quantization is set to float32 as well, and the sharder to shard the model is EmbeddingBagSharder.

In the third step (③), we design and implement a model generation component so that D<sup>3</sup> can automatically generate models. In this step, D<sup>3</sup> also generates model input that are valid for the corresponding models. Finally (④), D<sup>3</sup> executes the generated models and their input under the distributed settings to generate model output. Specifically, D<sup>3</sup> trains and evaluates a model  $M_i$  on input  $I_{ij}$  under a distributed setting  $S_k$ , which is denoted as  $TE(S_k, M_i, I_{ij})$ . We use  $O_{ijk}$  to denote the final evaluation output of model  $M_i$  trained on input  $I_{ij}$  under a distributed setting  $S_k$ , i.e.,  $O_{ijk} = TE(S_k, M_i, I_{ij})$ .

D<sup>3</sup> then compares the model output to detect inconsistency bugs. While our distributed rule enables us to detect hard-to-find inconsistency bugs, our model generation component may still expose crash bugs in DL software. Thus, D<sup>3</sup> also detects crash bugs when the evaluation of test scenarios crashes.

The rest of the approach section describes the equivalence rule definition (Section III-B), the distributed setting generation

TABLE I  
FUZZING OVERVIEW

	Fuzzing Parameter	Candidate Value
<b>Distributed Setting</b>	world size	{1, 2, 3, 4, 8}
	sharding type*	{TW, RW, CW, DP}
	device	{cpu, gpu}
	weight quantization	{int8, float32}
	activation quantization	{int8, float16*, float32}
	sharder type*	{EBS, EBCS}
<b>Model</b>	# of embedding bags	<b>EmbeddingBagCollection</b> {1, 5}
	# of embeddings	<b>Embedding Bag</b> {1, 1000}
	embedding dimension	{4, 8, 12, ..., 1,000}
	in dimension	<b>Linear</b> {1, 1,000}
	out dimension	{1, 1,000}

\*indicates PyTorch/TorchRec’s specific components and values. TW = table wise, RW = row wise, CW = column wise, DP = data parallel. EBS = EmbeddingBagSharder, EBCS = EmbeddingBagCollectionSharder.

(Section III-C), the model and model input generation (Section III-D), and the bug detection process (Section III-E).

### B. Distributed Equivalence Rule

To test deep to find inconsistency bugs, we create an equivalence rule for distributed deep learning libraries. We use the same definition of equivalence rules in the previous work [26], which defines specific conditions in deep learning libraries where different executions lead to equivalent output. For example, one EAGLE equivalence rule states that if a function has a sparse tensor version and a dense tensor version, the two versions should produce equivalent results. Otherwise, there are bugs in the implementations.

In this paper, we create a new *distributed equivalence rule*: **Distributed Equivalence Rule.** For any combination of the following distributed parameters: 1) **World Size**; 2) **Sharding Type**; 3) **Device**; 4) **Weight Quantization**; 5) **Activation Quantization**; and 6) **Sharder Type**.

*The distributed training and inference should produce output that is equivalent to that of the non-distributed training and inference counterparts.*

This rule implies that the output of distributed training and inference with two different distributed settings should also be equivalent.

Thus, a key task of our approach is to generate a large, diverse set of distributed settings and then compare model training and inference in these distributed settings. One distributed parameter is the world size, i.e., the number of devices or processes for distributed training. For example, distributed training on two GPUs versus four GPUs should produce equivalence output. Another distributed parameter is sharding type, i.e., how a model is divided into different devices or processes, e.g., row-wise sharding and column-wise sharding. For example, distributed training with row-wise sharding on two GPUs should produce equivalent output to that of distributed training with column-wise sharding on four GPUs.

### C. Generation of Distributed Settings

This section describes the set of distributed parameters and how we generate distributed settings following these parameters. Table I (Row ‘Distributed Setting’) presents the five distributed parameters and their possible values, while Section III-D ‘DLRM-like Model Generation’ describes Row ‘Model’ regarding model fuzzing.

**Distributed Parameter 1: World Size** World size is the number of processes participating in the distribution job. It is usually equal to the number of devices, such as the number of GPUs, in the distributed system. Rank is the unique ID given to a process so that the process can identify itself. For example, suppose a distributed system consists of four GPUs with each GPU running one process. Then in that system, the world size is four, and the ranks for the four processes are in [0, 1, 2, 3]. We compare the output of distributed training with one to eight world sizes. The World Size Equivalence rule applies to both DDP and DMP paradigms.

**Distributed Parameter 2: Sharding Type** Sharding is a concept in database systems that distributes a single database across multiple smaller databases, which can then be stored on multiple machines. The two common sharding types in database systems are horizontal sharding (each shard has the same schema but unique rows) and vertical sharding (each shard has a schema that is a proper subset of the original table’s schema). In distributed deep-learning systems, sharding is to split a model into multiple shards, where each shard is distributed to one processor. The sharding type defines the principle of splitting a deep learning model. Suppose we have a TorchRec model consisting of multiple embedding tables with each table for one feature. For example, table-wise sharding splits such a model by placing each table on one processor. While table-wise sharding keeps a whole embedding table on one processor, column-wise and row-wise sharding split an embedding table such that one table is placed on multiple processors. Column-wise sharding splits an embedding table by its embedding dimension and row-wise sharding splits the table by its first dimension. Data-parallel sharding is the same as DDP, which replicates the model on each processor. Data-parallel shards the dataset instead of the model.

**Distributed Parameter 3: Device** DL libraries usually provide support for both CPU and GPU devices. While libraries often have the same high-level API for users no matter whether CPUs or GPUs are used, they have different kernel implementations for the operations on different devices. For example, in PyTorch, there are operations that communicate between distributed processes, such as `all_gather` which gathers tensors from all the processes in a list and `all_reduce` which performs a reduce operations (e.g., reduce sum) to the tensor data across all machines in a way that all get the final result. Such operations are supported by the NCCL library when training or inference on GPUs while by the Gloo library when on CPUs. Although they have different implementations, we expect them to produce equivalent results when training or inferring the same model using the same model input.

**Distributed Parameter 4: Weight Quantization** Weight quantization refers to techniques for performing computations and storing tensors at lower bitwidths than floating point

precision. It can reduce the model size in storage as well as bandwidth requirements for the hardware platform. It can also speed up the model inference procedure. Although weight quantization changes the data type of the model’s parameters to lower precision, which inevitably leads to differences, the influence of quantization on to model’s prediction results should be similar between distributed settings and the non-distributed setting.

**Distributed Parameter 5: Activation Quantization** Besides weights, the model’s activations can also be quantized to low-precision data types to further reduce the memory cost and speed up communication between devices. Regardless of the activation data type used, the performance of the distributed model should be equivalent to that of the non-distributed version.

**Distributed Parameter 6: Sharder Type** The sharder implements partitions for the embedding tables according to the specified sharding type. Different types of sharders can shard different layers. For example, `EmbeddingBagSharder` shards `EmbeddingBag`, while `EmbeddingBagCollectionSharder` shards `EmbeddingBagCollection`, which is an optimized implementation of multiple embedding bags. Those sharders have different implementations, but they are expected to produce the same output, because the sharded models combined should be equivalent to the original model.

**Generation and Combination of Distributed Settings** We support the combination of different parameters. Distributed training and inference with a combination of values of five different distributed parameters should produce equivalent output to the non-distributed training and inference. For example, distributed training and inference with a mix of different world sizes and sharding types should produce equivalent output.

We first collect candidate values of the distributed parameters, by leveraging our domain knowledge about distributed DL and consulting the deep-learning libraries’ documentation. Table I lists all the parameters and their candidate values evaluated in this paper. The candidate values for the distributed parameter world size are {1, 2, 3, 4, 8}. The candidate values for sharding type are {table wise, row wise, column wise, data parallel}. The candidate values for the device are {cpu, gpu}. For weight quantization, the candidate values are int8, representing quantization is used and the model’s weights are quantized to int8, and float32, representing quantization is not used and the model’s weights are in their original data type float32. As for activation quantization, the candidate values are {int8, float16, float32}. For the sharder type, the candidate values are {`EmbeddingBagSharder`, `EmbeddingBagCollectionSharder`}, specifying how deep learning libraries shard models.

Then D<sup>3</sup> generates one candidate value for each distributed parameter in one distributed setting. For example, {world size: 1, sharding type: column wise, device: gpu, weight quantization: int8, activation quantization: float32, sharder type: `EmbeddingBagSharder`} is one distributed setting. D<sup>3</sup> generates distributed settings for all the combinations of the distributed parameters’ candidate values. Then we removed unsupported distributed settings by checking the DL library documentation and unit tests written by the developers.

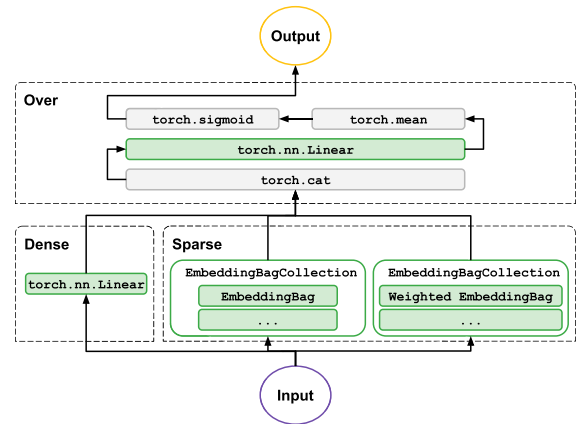


Fig. 3. DLRM-like model template. Green components denote model components on which D<sup>3</sup> fuzzes.

#### D. Generation of Models and Model Input

We start from popular, realistic DL structures and mutate them to generate a diverse set of models. We use (1) a DLRM-like structure to focus on fuzzing embedding components, and (2) a classic chain structure template and a cell-based structure template following previous work [46] to generate model structures to fuzz on other components.

First, Fig. 3 shows the template that we create for generating model structures. The template structure is a DLRM-like structure because our model template consists of the three components in DLRM: (1) a sparse component to process the sparse features that represent the categorical data, such as the rating of a movie, (2) a dense component to process the dense features, which usually represents the embedding of users, and (3) an over component that serves as an interaction among all the features. We choose a DLRM-like structure because DLRM is a real-world architecture, which makes it more likely to detect realistic bugs. We fuzz on all three components to mutate the DLRM-like model structure to generate a diverse set of models.

**DLRM-like Model Structure** The *sparse component* mainly consists of `EmbeddingBag` layers, which are used to handle the categorical features of the input. The `EmbeddingBag` layer is a kind of embedding layer. A general embedding layer is a lookup table that can convert the layer’s input to a fixed length of vectors. An `EmbeddingBag` layer is a general embedding layer followed by a sum/mean/max operation, while the `EmbeddingBag` is more efficient than using a chain of those operations because it does not need to initiate the intermediate embedding. An `EmbeddingBagCollection` layer collects multiple `EmbeddingBag` layers together so that the user only feeds one tensor to the `EmbeddingBagCollection` layer for all the embedding bags instead of one tensor for each embedding bag. We focus on the `EmbeddingBag` and `EmbeddingBagCollection` layers because they support distributed model parallelism with multiple sharding types, which is an important distributed setting for distributed deep-learning systems.

The *dense component* contains a single linear layer to process the dense features in the input. The *over component* first

concatenates the results from the dense component and sparse component, then applies a linear layer followed by mean and sigmoid functions to produce a score, which indicates how likely one would click their mouse given the input data.

**DLRM-like Model Generation** Table I Row ‘Model’ presents the details for DLRM-like model generation, which fuzzes all three components of the model structure. The sparse component consists of multiple embedding bags, with each embedding bag for one sparse feature of the input. We generate two types of embedding bags, i.e., weighted embedding bags and unweighted embedding bags. If per-sample weights are passed as arguments to embedding bags, they are called weighted embedding bags and the output of the embedding bags is scaled before performing a weighted reduction. Otherwise, for unweighted embedding bags, the output of the embedding bags will be directly reduced. In this project, we randomly generate 1–5 embedding bags and 1–5 weighted embedding bags.

For each embedding bag,  $D^3$  generates a random integer between 1 and 1,000 as the size of the dictionary of embeddings, and a random number that is a multiple of 4 between 1 and 1,000, as the size of embedding vectors. We use a multiple of 4 due to PyTorch’s FBGEMM library’s requirement.

For the dense component consisting of a linear layer,  $D^3$  randomly generates the shape of the linear layer. In other words, it generates the dimension of the linear layer matrix (in dimension, out dimension). The in dimension and out dimension each is an integer in the range of [1, 1,000].

For the over component, which also contains a dense layer,  $D^3$  randomly generates the out dimension between 1 and 1,000 inclusive. The in dimension is derived from the output dimension for all the input features, including the dense features and the sparse features from the dense and sparse components.

**Other Model Generation** Second, we follow the approach used in Muffin [46] to generate chain structure models and cell-based structure models. Muffin generates models by first generating the model structure and then generating each layer. For model structure generation, Muffin starts by selecting one template. Muffin implements two model templates. One is the chain structure with skip connections, which contains a sequence of layers with random skip connections. The other is the cell-based structure, which consists of a sequence of cells. Each cell is a DAG with one input vertex and one output vertex and each vertex in the DAG represents a DL layer. Given the generated structure information, then Muffin refines the layer information, i.e., determines the specific layer type for each vertex in the DAG.

**Layer Frozen Model Generation** Distributed training and quantization are implemented through conversions in PyTorch and TensorFlow. For instance, during distributed training in PyTorch, a single-device model (i.e., `torch.nn.Module`) is converted into a multiple-device distributed model (i.e., `torch.nn.parallel.DistributedDataParallel`). Similarly, in TensorFlow, quantization is achieved by converting a layer into a `QuantizeWrapper` layer. Besides adding new functionalities, such as data parallelism and quantization, these conversions should maintain the properties of the original layers, like whether a layer is trainable or not. However, bugs in the implementation can lead to incorrect conversions.

To verify the correctness of these conversions, we generate layer-frozen models by randomly freezing a layer in the model. This is done by setting `trainable = False` in TensorFlow and `requires_grad = False` in PyTorch. We then test these layer-frozen models using our distributed equivalence rule to check for any inconsistencies introduced by incorrect conversions.

**Model Input Generation** For each generated DL model,  $D^3$  generates model input according to the models’ input layers.

Since the model is generated according to the DLRM model template, the model input consists of three components: sparse features (i.e., the features for embedding bag layers), dense features (i.e., the features for fully connected layers), and labels. When  $D^3$  generates input,  $D^3$  reads parameters from the model’s input layers, e.g., the input dimension of the fully connected layers and the embedding layers. For example, if the model that  $D^3$  generated consists of one fully connected layer in its dense part and one embedding bag in its sparse part, the randomly generated input to this model consists of two tensors, one tensor with the shape of  $(n, i)$ , where  $n$  is the batch size and  $i$  is the input dimension of the fully connected layer, and one tensor with the shape of  $(n, r)$ , where  $r$  is a random number with value within the range  $[0, e)$ , where  $e$  is the vocabulary size of the embedding layer.

For the sparse features,  $D^3$  obtains the vocabulary size of each feature by reading certain parameters from each embedding bag, e.g. parameter `num_embeddings` of PyTorch’s layer API `torch.nn.EmbeddingBag`. The vocabulary sizes specify the range of input values for each feature. For example, if the `num_embeddings` is 100 for embedding bag  $eb_1$ , then the value of relative sparse feature  $f_1$  in the model input should be within the range of  $[0, 100)$ .

For dense features,  $D^3$  generates a tensor with the shape of  $[batch\_size, in\ dimension]$  and values between  $[0.0, 1.0]$ , where `batch_size` is set to 2,400 in our experiment and `in dimension` is the input dimension of the Linear layer.

For the labels,  $D^3$  generates a random vector with the shape of  $[batch\_size]$  and values between  $[0.0, 1.0]$ .

### E. Bug Detection

In this final step,  $D^3$  evaluates the generated test scenarios, i.e., the distributed settings generated in step two (Section III-C) and the models and model input generated in step three (Section III-D).  $D^3$  loads one pair of model  $M_i$  and model input  $I_{ij}$ , then trains  $M_i$  for a preset number of iterations under the generated distributed setting  $S_k$ . Then  $D^3$  uses the model for inference on  $I_{ij}$  to produce output  $O_{ijk}$ . That is, given a triple  $(S_k, M_i, I_{ij})$ ,  $D^3$  produces output  $O_{ijk}$ . After  $D^3$  evaluates all models and model input under different settings, it compares all  $O_{ijk}$  for the same  $i$  and  $j$  with different  $k$  values to detect inconsistencies. That is, comparing output values from the same model and model input trained and evaluated under all distributed and non-distributed settings.

To speed up the process, we focus on comparing distributed settings with non-distributed settings. For instance, with model  $M_1$  and input  $I_{11}$ , we compare  $O_{111}$ ,  $O_{112}$ , ..., with  $O_{110}$ , respectively, where  $S_0$  represents the non-distributed setting

and  $S_1, S_2, \dots$ , denote the distributed settings. Throughout the debugging process, we will further make comparisons between distributed settings to help debug and identify additional inconsistent setting pairs, which help developers fix bugs. D<sup>3</sup> reports inconsistent output, i.e., the element-wise difference of output vectors, that is bigger than the sum of two thresholds, one for absolute difference and one for relative difference, as potential inconsistency bugs (Section IV).

**Iteratively Mapping Inconsistencies to Bugs** Given the large number of inconsistencies, we design an iterative debugging approach to systematically map inconsistencies to bugs to help developers fix bugs. First, we cluster inconsistencies by the inconsistency-introducing APIs. Specifically, we employ the rate of change metric from CRADLE [25]. For each pair of distributed settings that produces an inconsistency above our thresholds, we calculate the Mean Absolute Distance (MAD) between each pair of corresponding hidden states from two executions of the same layer on two different test scenarios, i.e.,  $O_{ijk_1}$  and  $O_{ijk_2}$ . Then we calculate the rate of change for each layer, and finally cluster inconsistencies based on the layer API which produces the largest rate of change.

Second, we randomly sample test scenarios from each cluster for investigation. Upon identifying a bug, we seek a suitable fix. We check existing bug reports to find fixes for known bugs, such as converting standard batch normalization layers to synchronized batch normalization layers. For new bugs, we report the bugs to the developers to obtain a possible fix. If a viable fix is found, we apply it and subsequently re-execute the experiments that cause inconsistencies. After the re-execution, we count the number of inconsistencies again. The reduction in the number of inconsistencies is the number of inconsistencies resolved by the fix, which is counted as the number of inconsistencies caused by the bug. For the bugs that the developers have not fixed yet, i.e., the confirmed bugs and the reported bugs, we regard the inconsistencies in one cluster as one bug and wait for the developers' fixes. Once the developers fix the bug we reported, we apply the fix and resume the iterative debugging process. Ultimately, we obtain a list of bugs identified through this systematic debugging approach.

#### IV. EXPERIMENT SETUP

We tested PyTorch 1.12.0 (TorchRec 0.2.0) and TensorFlow 2.11.0. They were the latest versions of PyTorch and TensorFlow when we started building the tool (October 2022 for PyTorch and November 2022 for TensorFlow). The initial experiment was executed in February 2023 and an additional experiment was executed in June 2024 to obtain final results. We use docker to build environments.

We exclude a few distributed settings for PyTorch as they have not been supported by the library yet according to its documentation, e.g., the row-wise sharding on CPU is not supported by PyTorch.

Following previous work [26], we use the same inconsistency threshold formula that TensorFlow and PyTorch use in their test suite to determine whether the two output from the two models are equivalent. For example, model  $M_1$  and model input  $I_{11}$  are trained under distributed settings  $S_1$  and  $S_2$ , with respective output vectors  $O_{111}$  and  $O_{112}$ . Their output are equivalent if

TABLE II  
NUMBER OF INCONSISTENCIES FOUND BY D<sup>3</sup> AND DISTRIBUTED SETTINGS OF D<sup>3</sup>

DL Library	PyTorch	TensorFlow	Total
# of inconsistencies	10,478	5,595	16,073
# of distributed settings	77	24	101

the equation  $abs(O_{111} - O_{112}) \leq atol + rtol * abs(O_{112})$  is element-wise true, with  $atol = 5 * 10^{-4}$  and  $rtol = 1 * 10^{-4}$ . In the formula,  $atol$  is the threshold for absolute difference and  $rtol$  is for relative difference. We use both thresholds together to measure inconsistencies.

We use an Intel(R) Xeon(R) Gold 5220R server with 504GB memory, four NVIDIA RTX A5000 GPUs and four NVIDIA GeForce RTX 2080 Ti GPUs.

#### V. RESULTS

This section presents the results of our four Research Questions (RQs). RQ1 (Section V-A) presents the number of bugs D<sup>3</sup> detects. RQ2 (Section V-B) describes the bugs D<sup>3</sup> detected using those equivalence rules. RQ3 (Section V-C) compares D<sup>3</sup> to other DL-library testing techniques. RQ4 (Section V-D) studies D<sup>3</sup>'s execution time.

##### A. RQ1: How Many Bugs Does D<sup>3</sup> Detect?

We evaluate D<sup>3</sup> on two of the most popular distributed deep learning libraries, PyTorch and TensorFlow. For each library, D<sup>3</sup> generates 400 models and for each model, D<sup>3</sup> generates 10 model input, resulting in a total of 4,000 input generated. The same 4,000 input and 400 models are used in all 77 distributed settings for PyTorch, and another set of 4,000 input and 400 models generated for TensorFlow are used in all 24 distributed settings for TensorFlow. D<sup>3</sup> generates fewer distributed settings for TensorFlow because two distributed parameters are not supported in TensorFlow, i.e., sharding type and sharder type. Table II shows the total number of inconsistencies D<sup>3</sup> detects, and Table III describes the bugs D<sup>3</sup> detects.

Overall, D<sup>3</sup> detects 21 bugs, including 14 inconsistency bugs and seven crash bugs. Out of the 21 bugs, 12 are previously unknown bugs. 14 of the 21 bugs have been confirmed or fixed by developers. In Table III, we list root causes, affected distributed parameters, affected libraries, and the number of inconsistencies caused by each bug. The Status column shows whether the bugs are fixed, confirmed, or reported (waiting for replies from the developers). The New columns represent whether the bug is a new bug or a duplicate of a known bug. While D<sup>3</sup> detects crash bugs, of the 21 bugs detected by D<sup>3</sup>, 14 are inconsistency bugs, with eight being newly discovered, underscoring the significant contribution of the distributed equivalence rule.

Out of the 21 bugs, 11 are only detected by fuzzing the embedding components, demonstrating the usefulness of fuzzing embedding components in addition to other model structures. The remaining ten bugs are not specific to the embedding components and are detected by D<sup>3</sup>-generated models with chain and cell-base structures.

Although the distributed equivalence rule in D<sup>3</sup> is designed to detect inconsistency bugs by comparing the results from the



same models trained on the same model input under different distributed settings,  $D^3$  also effectively detects crash bugs due to the different distributed settings and the different model and model inputs  $D^3$  generates. For example,  $D^3$  detects a crash bug (Bug 15) when generating a distributed setting with the `EmbeddingBagSharder` sharder. This specific distributed setting, combined with a randomly generated model and inputs, led to a mismatch of length between the input and the model's embedding layer. This demonstrates  $D^3$ 's ability to detect crash bugs by generating varied distributed settings, models, and model input.

A single bug often causes many inconsistencies. Specifically, the 21 bugs that  $D^3$  detects map to 16,073 inconsistencies in TensorFlow and PyTorch. Table II shows the number of inconsistencies detected by  $D^3$ . Among the total 16,073 inconsistencies  $D^3$  detects, all inconsistencies indicate true bugs, with 366 inconsistencies mapped to Bug 1, 125 inconsistencies mapped to Bug 2, 1,323 inconsistencies mapped to Bug 3, 1,132 inconsistencies mapped to Bug 4, 944 inconsistencies mapped to Bug 5, 973 inconsistencies mapped to Bug 6, 230 inconsistencies mapped to Bug 7, 30 inconsistencies mapped to Bug 8, 3,947 inconsistencies mapped to Bug 9, 28 inconsistencies mapped to Bug 10, 408 inconsistencies mapped to Bug 11, 5,304 inconsistencies mapped to Bug 12, 1,147 inconsistencies mapped to Bug 13, and 116 inconsistencies mapped to Bug 14. For example, the 230 inconsistencies that map to Bug 7 result from 13 models that contain `BatchNormalization` layers. After we apply the fix, i.e., replacing `BatchNormalization` with `SyncBatchNormalization`, the number of inconsistencies decreases, indicating those reduced inconsistencies are caused by Bug 7.

To illustrate the effectiveness of the six distributed parameters, we investigate the inconsistency-triggering parameters for each bug. Inconsistency-triggering parameters are the parameters that when changed *alone* can cause inconsistencies or crashes. For example, denote the bug explained in Fig. 1 as bug 1.  $D^3$  detects inconsistencies caused by bug 1 between two settings,  $S_1$  and  $S_2$ , with  $S_1$  being {world size: 2, sharding type: column wise, device: gpu, weight quantization: float32, activation quantization: float32, sharder type: `EmbeddingBagCollectionSharder`} and  $S_2$  being {world size: 4, sharding type: column wise, device: gpu, weight quantization: float32, activation quantization: float32, sharder type: `EmbeddingBagCollectionSharder`}. The only difference between  $S_1$  and  $S_2$  is world size. If we detect inconsistencies between two distributed settings that have only one different parameter, that parameter is called an inconsistency-triggering parameter. In the above example, world size is an inconsistency-triggering parameter to bug 1.

Note that for bug 1, the sharding type is also an inconsistency-triggering parameter. Further investigation finds that the gradient aggregation setting only affects model parallelism (i.e., table wise, row wise, and column wise sharding). When the sharding type is set to data parallel, the distributed training will obtain correct results, i.e., using the average to aggregate per device gradients, which is inconsistent with other sharding types, e.g., row wise sharding. Therefore, sharding type is also an inconsistency-triggering parameter to bug 1.

Table III demonstrates that each of the six distributed parameters is an inconsistency-triggering parameter to at least one bug, indicating their effectiveness in distributed testing.

### B. RQ2: What Bugs Are Detected by $D^3$ ?

In this section, we describe the details of the bugs that  $D^3$  detects in addition to the bug in Fig. 1.

*a) Bug 1:* (PyTorch gradient aggregation bug) This is the bug that Section I-B and Fig. 1 describe.

In principle, gradient aggregation in distributed training should be consistent with the loss function. For example, when using mean squared error (MSE) as the loss function in the training process, the gradient computed is the per-sample average gradient. When switching to DDP in this case, in order to obtain the same training result as in the non-distributed setting, users need to use the average to aggregate per-device gradients to get the same gradients as those in the non-distributed setting.

Making sure the gradient calculation is the same is important, especially in the product development process. This is because training hyperparameters should correspond to gradients. For example, in Fig. 1, the gradient becomes smaller when changing from the distributed setting to the non-distributed setting. If the same learning rate is used, the step size in each iteration's optimization is smaller, finally leading to accuracy differences. Hyperparameters are usually fine-tuned for the best training performance. The fine-tuning process is expensive and highly relies on human expertise. In order to avoid repeating hyperparameters fine-tuning, it is essential to make gradient calculation the same as the non-distributed process, so that the same hyperparameters can be used for training with any world size.

*b) Bug 2:* (TensorFlow Keras distributed layer bug)  $D^3$  detects inconsistencies when training a TensorFlow model consisting of Keras layers with different world sizes. The TensorFlow developers confirmed this is an issue with the Keras layers. This bug has been fixed in the latest Keras nightly version after we reported it.

*c) Bug 3:* (PyTorch quantized weighted `EmbeddingBagCollection` bug):  $D^3$  detects inconsistencies when training DLRM-like models with weight quantization and NCCL backend under different world sizes. Further investigation shows the model's weighted `EmbeddingBagCollection` layer's output have huge differences under the two distributed settings (e.g., between training on one GPU and training on eight GPUs), which could be the cause for this inconsistency. These inconsistencies have been fixed in the recent nightly versions.

*d) Bug 4:* (PyTorch `BatchNorm2d` bug):  $D^3$  detects inconsistencies when training a model containing `torch.nn.BatchNorm2d` layers using different world sizes, e.g., training on one GPU versus training on eight GPUs. The bug is caused by the lack of synchronization across devices with regard to `BatchNorm2d` layers. Specifically, the replicated `BatchNorm2d` layers on each device are trained using their local batches which are different across devices. This leads to different weights in different devices. This bug severely affects model accuracy, leading to a drop in model accuracy when training on multiple GPUs. We confirmed that this bug is a known

TABLE III

BUGS FOUND BY D<sup>3</sup>. BUG 1 IS THE BUG IN FIG. 1. “# INCONSISTENCIES” REPRESENTS THE NUMBER OF INCONSISTENCIES DETECTED FOR EACH BUG. “-” INDICATES A CRASH BUG. D<sup>3</sup> DETECTS 21 BUGS, 14 OF WHICH ARE INCONSISTENCY BUGS. MAJORITY (14) OF THE 21 BUGS ARE CONFIRMED OR FIXED BY THE DEVELOPERS. MOST (12) OF THE 21 BUGS ARE PREVIOUSLY UNKNOWN BUGS

Bug ID	Root Cause	Inconsistency-Triggering Parameters	Software	# Inconsistencies	Status	New
1	Gradient aggregation	world size & sharding type	PyTorch	366	fixed	yes
2	Keras distributed layer	world size	TensorFlow	125	fixed	yes
3	Quantized weighted EBC	world size & weight quant	PyTorch	1,323	fixed	no
4	Batch normalization	world size	PyTorch	1,132	fixed	no
5	Integer activation quantization	world size & weight quant & activation quant	PyTorch	944	fixed	no
6	Float activation quantization	world size & weight quant & activation quant	PyTorch	973	fixed	no
7	Batch normalization	world size	TensorFlow	230	fixed	no
8	XLA precision error	world size & device	TensorFlow	30	fixed	no
9	Quantize apply	world size & weight quant	TensorFlow	3,947	confirmed	yes
10	Synchronized batch normalization	world size & device	PyTorch	28	reported	yes
11	NaN results	world size & weight quant	PyTorch	408	reported	yes
12	NaN results	world size & weight quant & activation quant	PyTorch	5,304	reported	yes
13	Synchronized batch normalization	world size & device	TensorFlow	1,147	reported	yes
14	Quantization trainable=False	world size & weight quant	TensorFlow	116	reported	yes
15	Dummy feature name	sharder type	PyTorch	-	fixed	no
16	Key mismatch error	weight quant	PyTorch	-	fixed	no
17	CUDA error	device	PyTorch	-	fixed	no
18	Missing configuration	sharder type	PyTorch	-	confirmed	yes
19	MirroredStrategy overhead	world size & device	TensorFlow	-	confirmed	yes
20	CUDA internal assert failed	device & weight quant	PyTorch	-	reported	yes
21	to_dict() error	weight quant & activation quant	PyTorch	-	reported	yes

bug (<https://github.com/pytorch/pytorch/issues/2584>) raised by previous users, and the developers added a synchronized version API `torch.nn.SyncBatchNorm` to fix this bug.

*e) Bug 5&6:* (PyTorch activation quantization bug): D<sup>3</sup> detects two inconsistency bugs during model inferencing when quantizing the activation to integer (e.g., `torch.qint8`) and float (e.g., `torch.float16`), respectively. Both bugs cause the weighted sparse layer to produce large inconsistencies during inferencing, which then propagate to the model’s final output. However, with activation quantized to `torch.float16`, there are large inconsistencies in the sparse layer’s output as well, while the sparse layer’s output is identical when the activation is quantized to `torch.qint8`. Both bugs have been fixed in the latest version of TorchRec.

*f) Bug 7:* (TensorFlow BatchNormalization bug): D<sup>3</sup> detects inconsistencies when training a model containing `tf.keras.layers.BatchNormalization` layer under different world sizes, e.g., training on 1 GPU versus training on 8 GPUs. This bug has the same cause as bug 2. After the previous users submitted a report about this bug (<https://github.com/pytorch/pytorch/issues/2584>), the developers fixed it by providing a synchronized version API `tf.keras.layers.experimental.SyncBatchNormalization` which applies batch normalization to the global batches.

*g) Bug 8:* (TensorFlow XLA precision bug): D<sup>3</sup> detects inconsistencies when training a TensorFlow model with CPU backends and GPU backends. The developer confirmed the inconsistencies are precision-related because of XLA fusion. This bug has been fixed in the latest version of TensorFlow.

*h) Bug 9:* (TensorFlow `quantize_apply` bug): D<sup>3</sup> detects inconsistencies when training a quantize-aware model converted using `tfmot.quantization.keras.quantize_apply` under different world sizes, e.g., training

on one GPU versus on two GPUs. There are no inconsistencies when the model is trained under the same world sizes, e.g., training on two CPUs versus on two GPUs. After we report the bug, the developers confirm the root cause is that during quantization, the min and max value of each activation is not synchronized over distributed units. Many inconsistencies are related to this bug because it affects all quantization experiments.

*i) Bug 10:* (PyTorch `SyncBatchNorm` bug) D<sup>3</sup> detects inconsistencies with specific models and input even after converting the `BatchNorm2d` layers to `SyncBatchNorm`. This bug happens when training the model with multiple GPUs. We have reported the bug to the developers.

*j) Bug 11:* (PyTorch training NaN results bug): D<sup>3</sup> detects NaN when training a quantized model with NCCL backend and world size setting to 4. The model’s weighted `EmbeddingBagCollection` layer produces some NaN values in its output, which in turn causes the subsequent layer to produce all NaN output.

*k) Bug 12:* (PyTorch inferencing NaN bug): D<sup>3</sup> detects another NaN bug when inferencing a model with activation quantized to `torch.qint8`. Further investigation reveals that the sparse layer and the weighted sparse layer produce NaN output, which then propagates to the model’s output. This bug is different from Bug 11 because reproducing this bug requires only inferencing, while Bug 11 is caused by training. We reproduced the bug with the latest TorchRec and reported it to the developers.

*l) Bug 13:* (TensorFlow `SyncBatchNormalization` bug): D<sup>3</sup> detects inconsistencies when training a model containing `SyncBatchNormalization` under different world sizes on CPU devices. However, there are no inconsistencies when the same model is trained on the same input on different numbers of GPUs. Thus, D<sup>3</sup> successfully detects this bug in

the `SyncBatchNormalization` layer in a recent version of TensorFlow 2.11.0. However, this layer API is already deprecated in the latest version of TensorFlow 2.12.0 which is only a few months more recent. But we found that the same bug exists in its replacement `BatchNormalization` layer when `synchronized` is set to `true`.

*m) Bug 14:* (TensorFlow `trainable=False` bug):  $D^3$  detects inconsistencies when training a TensorFlow model containing a quantized dense layer with `trainable` set to `False`. Further investigation reveals that when setting `trainable=False`, the kernel of the quantized dense layer is still trainable. There is a bug in the quantization conversion function that does not correctly set the `trainable` property from the original dense layer thus leading to inconsistent results after training.

*n) Bug 15:* (PyTorch `EmbeddingBagSharder` dummy feature name bug): When `EmbeddingBagSharder` shards the embedding bags, it generates a dummy feature name and assigns it to all sharded embedding bags, causing the sharded embedding bags to have the same feature name. We find that when testing with some specific model that  $D^3$  generates, the sharded embedding bags that have the same feature name are regarded as one embedding bag, which causes a mismatch of length between the input and the model's embedding layer. This triggers an assertion in TorchRec's source code.

*o) Bug 16:* (PyTorch `quantize_embeddings` bug): TorchRec raises a key mismatch error when saving and loading the `state_dict` of a model quantized using `quantize_embeddings`. The model's `state_dict`, which is a dictionary that maps model parameters' names with their values, alters after quantization. This is caused by inconsistencies between parameter names in FBGEMM backend kernels and the canonical `EmbeddingBag` representation. The developers fix this bug by adding a mapping to transfer parameter names in backend kernels to the canonical representation.

*p) Bug 17:* (PyTorch `DataLoader` bug): A CUDA error occurs when training a TorchRec model using `DataLoader` with multiple workers on GPU. `DataLoader` is a class provided by PyTorch that has many options for data loading. `num_workers` is an option to enable multiprocessing for `DataLoader`. By default, `num_workers` is set to 0, which represents single process data loading. When `num_workers` is set to a positive number, `DataLoader` creates `num_workers` subprocesses to speed up data loading. However, a bug occurred when using `DataLoader` with `num_workers` equals two to generate CUDA tensors to train a DLRM model. The developers fixed it by disabling multiprocessing for `DataLoader`.

*q) Bug 18:* (PyTorch `EmbeddingBagSharder` missing configuration bug): This bug happens in TorchRec when specifying `EmbeddingBagSharder` to shard a model containing `EmbeddingBagCollection`. The sharder tries to shard the model table-wise, but the table-wise sharding is not implemented, so it raises an error. In TorchRec, a planner is called to generate an optimized sharding plan for a given module with its shardable parameters according to the provided sharders, the topology of the devices, and any customized constraints specified by users. The planner first

generates all possible sharding plans by enumerating all combinations of available sharding types and computing kernels and then searches for an optimized sharding plan. The available sharding types and compute kernels are defined in the sharder's class. However, `EmbeddingBagSharder` doesn't support table-wise sharding while it still includes the table-wise option in its available sharding types, which leads to this error.

*r) Bug 19:* (TensorFlow `MirroredStrategy` overhead bug):  $D^3$  detects a hang that occurs when using `MirroredStrategy` with eager mode on more than one GPU which does not happen with graph mode or on CPUs. It is due to significant overhead when using `MirroredStrategy` in eager mode with multiple GPUs. This bug prevents the users from utilizing the benefit of eager mode to debug efficiently.

*s) Bug 20:* (PyTorch quantization Gloo backend bug): The Gloo and NCCL backends are the two collective communications libraries for distributed training of DL models mostly on CPUs and GPUs, respectively.  $D^3$  detects crashes when training quantized models with this Gloo backend. This bug occurs exclusively when training quantized models with the Gloo backend and not with the NCCL backend.

*t) Bug 21:* (PyTorch `to_dict()` crash bug): A crash bug occurs when transferring the embedding bag collection layer output to a dictionary in a DLRM model with activation quantized to `torch.qint8`. The output of the embedding bag collection layer, a concatenated tensor of each embedding bag's output, should be converted into a dictionary using `length_per_key` to determine each embedding bag output's length pre-concatenation. However, TorchRec raises an error, claiming the sum of `length_per_key` does not match the concatenated tensor length. This discrepancy makes the tensor invalid, preventing access to individual embedding bag output.

**Case study** In this section, we use Bug 5 as an example to illustrate how  $D^3$  identifies a bug. After  $D^3$  completes the experiment, we compare the final evaluation output between distributed and non-distributed settings to detect inconsistencies. We then run a clustering algorithm to group inconsistencies based on the rate of change metric. As a result, 944 inconsistencies are clustered within the weighted `EmbeddingBagCollection` layers.

Next, we randomly sampled five inconsistencies and discovered that all exhibited significant inconsistencies in the weighted `EmbeddingBagCollection` layers during a single forward pass without training. We created a minimal reproduction program based on the detected bug pattern. Before submitting a bug report to the developers, we ran the reproduction program on the latest versions of PyTorch and TorchRec (torch 2.3.0, torchrec 0.7.0) and found that the reproduction program did not produce inconsistencies, indicating that the bugs had been fixed by the developers.

Since this bug was fixed silently and we could not identify the exact commit that resolved the issue, we considered updating to the latest version as the fix for this bug. Finally, we applied the fix to all inconsistencies in the cluster by rerunning all test scenarios with the latest versions to confirm that all inconsistencies were resolved.

TABLE IV  
ANALYSIS QUESTION RESULTS FOR THE BUGS FOUND BY D<sup>3</sup>. ✓ REPRESENTS DETECTING THIS BUG REQUIRES THE SPECIFIC FEATURE IN THE ANALYSIS QUESTION, WHICH INDICATES EXISTING METHODS WOULD FAIL TO DETECT THIS BUG

Analysis Questions	Bug ID																				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
<b>QI) Distributed setting or rule</b>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
<b>QII) Multi-layer model</b>	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓
<b>QIII) Embedding bag layer</b>	✓		✓		✓	✓				✓	✓	✓			✓	✓		✓		✓	✓
<b>QIV) Training</b>	✓	✓	✓	✓			✓	✓	✓	✓	✓		✓						✓		✓

### C. RQ3: Does D<sup>3</sup> Detect Bugs That Existing DL-Library Testing Techniques Cannot Find?

In this section, we compare D<sup>3</sup> with existing techniques that differential test DL libraries to study the contributions of each component of D<sup>3</sup>. The main contributions of D<sup>3</sup> include 1) the distributed settings generation and the distributed equivalence rule, 2) the generation of multi-layer DL models 3) the generation of DL models with embedding bag layers, and 4) testing the training phase as opposed to testing the inference phase only like CRADLE [25], Audee [27], and LEMON [28]. We qualitatively analyze each bug found by D<sup>3</sup> and answer the questions below: **QI**) Does detecting this bug require a distributed setting or a distributed equivalence rule? **QII**) Does detecting this bug require a *multi-layer* DL model? **QIII**) Does detecting this bug require a DL model with *embedding bag layers*? **QIV**) Does detecting this bug require at least one step of training?

We first study the four analysis questions **QI** to **QIV**, and then use those answers to quantify, how many bugs existing DL-library testing techniques cannot detect out of the 21 bugs that D<sup>3</sup> detects. *Regarding QI, out of the total of 21 bugs detected by D<sup>3</sup>, 20 bugs require setting up distributed settings or a distributed equivalence rule as the oracle to detect.* For example, to detect Bug 5, it requires a distributed setting with world size greater than one, and both weight and activation quantization set to int8. It also requires comparing the results from the distributed setting and the non-distributed setting to detect inconsistencies. D<sup>3</sup> is the first approach that defines a distributed equivalence rule and generates distributed settings to test distributed DL software. Since none of the prior DL-library testing techniques generates distributed settings, they fail to detect those bugs.

To illustrate D<sup>3</sup>'s contributions in DL model generation (**QII** & **QIII**) and the testing of model training (**QIV**), we compare D<sup>3</sup> with previous approaches with distributed settings added. In other words, we study if we add D<sup>3</sup>'s distributed setting generation and equivalence rules to existing approaches, whether the enhanced existing approaches can detect the bugs that D<sup>3</sup> detects. *Regarding QII, out of the total 21 bugs, detecting 17 bugs requires multi-layer DL models that D<sup>3</sup> generates. As for QIII, 11 out of the total 21 bugs need models with embedding bag layers to detect. For QIV, 11 out of the 21 require at least one step of training to trigger.*

Table IV summarizes the bugs that existing DL-library testing tools cannot detect. We compare D<sup>3</sup> with fifteen existing tools EAGLE [26], DocTor [47], FreeFuzz [29],  $\nabla$ Fuzz [48],

TABLE V  
EXECUTION TIME OF D<sup>3</sup>

	PyTorch	TensorFlow
# (model, model input) generated	4,000	4,000
Time per pair (seconds)	373	119

FuzzGPT [49], TitanFuzz [50], DeepREL [30], CRADLE [25], Audee [27], LEMON [28], NeuRI [51], Muffin [46], Ramos [52], GenCoG [53], and HirGen [54]. API-based DL-library testing approaches, i.e., EAGLE, DocTor, FreeFuzz,  $\nabla$ Fuzz, FuzzGPT, TitanFuzz, and DeepREL, cannot find 17 of the bugs detected by D<sup>3</sup> because they cannot generate multi-layer models (**QII**). Muffin hardcoded the supported layer for generation which does not contain the embedding bag layer. Therefore, it cannot detect 11 bugs that require models with embedding bag layers as test inputs (**QIII**). NeuRI automatically collects layers from developer test cases, however, it focuses on testing the inference stage of DL models so it cannot detect the 11 bugs that require DL training (**QIV**). The remaining approaches, i.e., CRADLE, Audee, LEMON, Ramos, GenCoG, and HirGen, do not generate embedding bag layers (**QIII**) and only test the inference stage (**QIV**). They cannot detect 19 of the bugs.

### D. RQ4: What Is the Run Time of D<sup>3</sup>?

Table V shows D<sup>3</sup>'s execution time. On average, it takes 373 seconds to evaluate a pair of (model, model input) on our equivalence rules in PyTorch and 119 seconds in TensorFlow.

## VI. THREATS TO VALIDITY

**D<sup>3</sup> does not find all bugs** Since we use a threshold to define inconsistencies, we might miss bugs that cause very small differences in the prediction results. To mitigate this threat, we use a threshold used by popular DL libraries to measure the inconsistencies. As a result, D<sup>3</sup> is effective in detecting 21 bugs in PyTorch/TorchRec and TensorFlow automatically.

**Generalizability** D<sup>3</sup>'s generalizability to different DL libraries with a variety of DL model types has not been measured beyond the two libraries evaluated. Also, the generality of the distributed equivalence rule and the model generation was not quantified beyond PyTorch and TensorFlow. However, D<sup>3</sup> detects 21 bugs, including 14 inconsistency bugs, across two of the most popular DL libraries, i.e., PyTorch/TorchRec and TensorFlow. This already demonstrates D<sup>3</sup>'s capabilities of finding bugs in different distributed DL libraries. In addition,

$D^3$  applies the equivalence rules to multiple model templates, including the DLRM-like models, chain structure models, and cell-based structure models, which cover a diverse set of DL model types. It is straightforward to apply  $D^3$ 's equivalence rule on other model templates. Finally,  $D^3$  provided a new DL model template for the DLRM-like models, which could be used to enhance existing DL model generation tools, such as Muffin.

**Nondeterminism** Not all inconsistencies are bugs because DL model training can be nondeterministic [43]. We mitigate nondeterminism by using the same random seed to make the model training procedure algorithmically reproducible. We also use a threshold used by popular DL libraries to take into consideration floating-point precision inconsistencies. We adopt one-step training to minimize the nondeterminism of DL model training. Overall, all of the total 16,073 inconsistencies that  $D^3$  detects indicate true bugs.

## VII. RELATED WORK

**Differential testing of DL libraries**  $D^3$  is closely related to EAGLE [26] which applies differential testing using equivalent computational graphs to test a single DL library. EAGLE uses equivalent graphs which use different Application Programming Interfaces (APIs), data types, or optimizations to achieve the same functionality.  $D^3$  focuses on testing distributed DL libraries whereas none of the 16 equivalent rules proposed by EAGLE can detect bugs in distributed DL training code.

Some recent work also leverages differential testing by comparing results between CPU and GPU runs [29] or between automatically matched equivalent DL APIs [30] to detect inconsistency bugs. Unlike these approaches,  $D^3$  detects bugs in the DL-distributed training code with its distributed equivalent rule.

Other work [25], [27], [28], [52], [55], [56], [57], [58], [59], [60] also uses differential testing to find inconsistencies between DL libraries. 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, Keras 2 (used in [25], [27], [28], [59]) no longer supports multiple backends. The new Keras 3 supports three DL backends, i.e., JAX, TensorFlow, and PyTorch. However, cross-checking different libraries is not possible for distributed parameters that only exist in one DL library. For example, sharder types and sharding types are two distributed parameters that only exist in PyTorch/TorchRec but not in TensorFlow. It is not possible to detect bugs caused by the two distributed parameters by cross-checking different DL libraries. One could use MMDnn [61] or ONNX [62] to transfer models across DL libraries, however only a few popular layers are supported by MMDnn (e.g., RNN layers are not supported) and one of the most popular DL libraries, PyTorch, cannot execute ONNX models. Srisakaokul et al. [57] only reimplements two ML algorithms (K-Nearest Neighbours and Naive Bayes) when using differential testing on Weka, Rapid Miner, and KNIME. Ramos [52] summarizes the API mapping rule for model initialization methods. Gandalf [60] adopts the context-free grammar and designed a series of equivalent metamorphic relationships to generate equivalent models

in different DL libraries. However, those papers implement a subset of DL computation in different frameworks and require heavy engineering. In contrast, similar to EAGLE,  $D^3$  uses the equivalence rule to find bugs in DL frameworks, which is not limited by third-party libraries (converter or high-level API support).

**Fuzzing DL libraries** Another popular approach to testing DL libraries is fuzzing. Classic fuzzing techniques [63], [64], [65] find some crash bugs, while more DL-specific fuzzing techniques have been proposed [47], [66], [67], [68]. However, they only focus on detecting crashes and testing API-level functions.

Recent work fuzzes [46], [69], [70] or generates [28], [71] DL models to test DL libraries. However, no prior work generates distributed settings to test distributed DL software.  $D^3$  applies DL model generation using multiple model templates along with the distributed differential testing rule to test DL libraries code that handles distributed computation.

**Other work testing DL libraries** Static analysis has been used to detect specific types of bugs (e.g., shape-related bugs) in DL systems [72].  $D^3$  finds very diverse bugs in DL systems (Section V-B) that are hard to find without equivalent distributed settings and model generation. Metamorphic testing has also been used to test DL compilers [73] which focuses on finding bugs at the lower level in DL compilers. Other work also applies metamorphic testing to validate ML classifiers [74], [75], [76], [77]. 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-world bugs [78].

**Differential Testing of DL Models** Prior work [21], [22], [23], [24] applies differential testing to test the trained DL models (i.e., the trained weights) instead of the underlying DL libraries that implement machine learning algorithms. For example, DeepXplore [22] introduces neuron coverage to measure testing coverage in CNN models and guide test input generation or OGMA [23] adapts a grammar-based input generation method to test NLP models. These approaches are orthogonal to our work because they test the correctness of DL models, while we test the correctness of DL libraries, i.e., *software implementations* of models. These prior techniques are not designed to detect bugs in DL libraries, because they compare the output of *similar* DL models to detect model bugs, which manifest by input instances that make these models generate incorrect output. On the other hand, our work focuses on comparing the output of the *same* model under different distributed settings to detect library implementation bugs. Existing work addresses neither the challenge of identifying equivalent distributed settings, nor the challenge of testing distributed DL training. For the latter, the bug in Fig. 1 is hard to detect for multiple reasons including generating the specific sharding scheme (Section I-B) for example.

**Differential testing for compilers** Differential testing has been used for testing compilers [79], [80], [81], [82]. 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 output for other input. Our work is different since program compilation is a different problem than DL graph execution which presents its own challenges.

## VIII. CONCLUSION

We propose D<sup>3</sup>, a new differential testing approach that uses distributed equivalence rule and model generation to test distributed deep learning software. We collected and fuzzed six distributed parameters that can generate equivalent distributed settings under which the same model and model input trained should produce equivalent prediction results. We evaluated D<sup>3</sup> on the two most popular DL libraries, PyTorch/TorchRec and TensorFlow, and found 21 bugs, 12 of which are previously unknown bugs. Future work includes extending our approach to fuzz other distributed components e.g., the cluster setup, to detect more types of bugs such as configuration bugs in the distributed deep learning software.

## REFERENCES

- [1] X. Yi et al., Eds., Sampling-bias-corrected neural modeling large corpus item recommendations, in *Proc. 13th ACM Conf. Recommender Syst.*, 2019, pp. 269–277.
- [2] M. Naumov et al., “Deep Learning recommendation model for personalization and recommendation Systems,” 2019, *arXiv:1906.00091*.
- [3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “DeepDriving: Learning affordance for direct perception in autonomous driving,” in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2015, pp. 2722–2730.
- [4] M. Popel, M. Tomkova, and J. Tomek, “Transforming machine translation: A deep learning system reaches news translation quality comparable to human professionals,” *Nat. Commun.*, vol. 11, no. 1, p. 4381, 2020.
- [5] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “CoCoNuT: Combining context-aware neural translation models using ensemble for program repair,” in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 101–114.
- [6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” 2019, *arXiv:1810.04805*. [Online]. Available: <https://arxiv.org/abs/1810.04805>
- [7] T. Brown et al., “Language models are few-shot learners,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., Red Hook, NY, USA: Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>
- [8] D. Narayanan et al., “Efficient large-scale language model training on GPU clusters using megatron-LM,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA: ACM, 2021, doi: 10.1145/3458817.3476209.
- [9] J. Wei, X. Zhang, Z. Ji, J. Li, and Z. Wei, “Deploying and scaling distributed parallel deep neural networks on the tianhe-3 prototype system,” *Sci. Rep.*, vol. 11, no. 1, 2021, Art. no. 20244, doi: 10.1038/s41598-021-98794-z.
- [10] Y. Jiang, F. Fu, X. Miao, X. Nie, and B. Cui, “OSDP: Optimal sharded data parallel for distributed deep learning,” in *Proc. 32nd Int. Joint Conf. Artif. Intell. (IJCAI)*, E. Elkind, Ed. 2023, pp. 2142–2150, doi: 10.24963/ijcai.2023/238.
- [11] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” *ACM Comput. Surv.*, vol. 52, no. 4, pp. 65:1–65:43, Aug. 2019, doi: <https://doi.org/10.1145/3320060>.
- [12] “TensorFlow GitHub issues,” TensorFlow. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/tensorflow/tensorflow/issues>
- [13] “TensorFlow GitHub pull requests,” TensorFlow. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/tensorflow/tensorflow/pulls>
- [14] “PyTorch GitHub issues,” PyTorch. Accessed: 2022. [Online]. Available: <https://github.com/pytorch/pytorch/issues>
- [15] “PyTorch GitHub pull requests,” PyTorch. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/pytorch/pytorch/pulls>
- [16] “TorchRec GitHub issues,” TorchRec. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/pytorch/torchrec/issues>
- [17] “TorchRec GitHub pull requests,” TorchRec. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/pytorch/torchrec/pulls>
- [18] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, “An empirical study on TensorFlow program bugs,” in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2018, pp. 129–140, doi: 10.1145/3213846.3213866.
- [19] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2019, pp. 510–520, doi: 10.1145/3338906.3338955
- [20] Y. Yang, T. He, Z. Xia, and Y. Feng, “A comprehensive empirical study on bug characteristics of deep learning frameworks,” *Inf. Softw. Technol.*, vol. 151, 2022, Art. no. 107004. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584922001306>
- [21] J. Guo, Y. Zhao, H. Song, and Y. Jiang, “Coverage guided differential adversarial testing of deep learning systems,” *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 933–942, Apr./Jun. 2021.
- [22] K. Pei, Y. Cao, J. Yang, and S. Jana, “DeepXplore: Automated whitebox testing of deep learning systems,” *Commun. ACM*, vol. 62, no. 11, pp. 137–145, Oct. 2019, doi: 10.1145/3361566.
- [23] S. Udeshi and S. Chattopadhyay, “Grammar based directed testing of machine learning systems,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2487–2503, Feb. 2019.
- [24] Y. Tian, K. Pei, S. Jana, and B. Ray, “DeepTest: Automated testing of deep-neural-network-driven autonomous cars,” in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2018, pp. 303–314, doi: 10.1145/3180155.3180220
- [25] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, “CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 1027–1038.
- [26] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan, “EAGLE: Creating equivalent graphs to test deep learning libraries,” in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 798–810.
- [27] Q. Guo et al., “Audee: Automated testing for deep learning frameworks,” in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2020, pp. 486–498.
- [28] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, “Deep learning library testing via effective model generation,” in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: ACM, 2020, pp. 788–799.
- [29] A. Wei, Y. Deng, C. Yang, and L. Zhang, “Free lunch for testing: Fuzzing deep-learning libraries from open source,” in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 995–1007, doi: 10.1145/3510003.3510041.
- [30] Y. Deng, C. Yang, A. Wei, and L. Zhang, “Fuzzing deep-learning libraries via automated relational API inference,” *Proc. 30th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA: Association for Computing Machinery, 2022, p. 44–56. [Online]. Available: <https://doi.org/10.1145/3540250.3549085>
- [31] A. Paszke et al., “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. Adv. Neural Inf. Process. Syst.* 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., Red Hook, NY, USA: Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [32] M. Abadi et al., “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Accessed: Sep. 9, 2022. [Online]. Available: <https://www.tensorflow.org/>
- [33] “TorchRec,” TorchRec. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/pytorch/torchrec>
- [34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: <http://www.deeplearningbook.org>
- [35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
- [36] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
- [37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
- [38] J. Dean et al., “Large scale distributed deep networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Red Hook, NY, USA: Curran Associates, Inc., 2012. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf)
- [39] M. Zinkevich, M. Weimer, L. Li, and A. Smola, “Parallelized stochastic gradient descent,” in *Proc. Adv. Neural Inf. Process. Syst.*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Red Hook, NY, USA: Curran Associates, Inc., 2010. [On-

- line]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2010/file/abea47ba24142ed16b7d8fbf2c740e0d-Paper.pdf)
- [40] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Human Lang. Technol.: Annu. Conf. North Am. Chapter Assoc. Comput. Linguistics*, Dec. 2010, pp. 456–464.
- [41] Y. Huang et al., *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [42] S. Fan et al., "DAPPLE: A pipelined data parallel approach for training large models," in *Proc. 26th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, New York, NY, USA: ACM, 2021, pp. 431–445, doi: 10.1145/3437801.3441593.
- [43] H. V. Pham et al., "Problems and opportunities in training deep learning software systems: An analysis of variance," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, New York, NY, USA: ACM, 2021, pp. 771–783, doi: 10.1145/3324884.3416545.
- [44] "Pytorch reproducibility," PyTorch. Accessed: 2022. [Online]. Available: <https://pytorch.org/docs/stable/notes/randomness.html#reproducibility>
- [45] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "DiffStream: Differential output testing for stream processing programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020, p. 153:1–153:29, Nov. 2020. [Online]. Available: <https://doi.org/10.1145/3428221>
- [46] J. Gu, X. Luo, Y. Zhou, and X. Wang, "Muffin: Testing deep learning libraries via neural architecture fuzzing," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2022, pp. 1418–1430, doi: 10.1145/3510003.3510092.
- [47] D. Xie et al., "DocTer: Documentation-guided fuzzing for testing deep learning API functions," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal., (ISSTA)*, New York, NY, USA: ACM, 2022, pp. 176–188, doi: 10.1145/3533767.3534220.
- [48] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in *Proc. 45th Int. Conf. Softw. Eng., (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1174–1186, doi: 10.1109/ICSE48619.2023.00105.
- [49] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, pp. 1–13, doi: 10.1145/3597503.3623343.
- [50] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2023, pp. 423–435, doi: 10.1145/3597926.3598067.
- [51] J. Liu, J. Peng, Y. Wang, and L. Zhang, "NeuRI: Diversifying DNN generation via inductive rule inference," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2023, pp. 657–669, doi: 10.1145/3611643.3616337.
- [52] Y. Zou, H. Sun, C. Fang, J. Liu, and Z. Zhang, "Deep learning framework testing via hierarchical and heuristic model generation," *J. Syst. Softw.*, vol. 201, 2023, Art. no. 111681. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121223000766>
- [53] Z. Wang et al., "GenCoG: A DSL-based approach to generating computation graphs for TVM testing," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2023, pp. 904–916, doi: 10.1145/3597926.3598105.
- [54] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2023, pp. 248–260, doi: 10.1145/3597926.3598053.
- [55] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 1–36, Jan. 2022.
- [56] S. Dutta, O. Legunzen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 574–586.
- [57] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Proc. Workshops 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 384–391.
- [58] J. Vanover, X. Deng, and C. Rubio-González, "Discovering discrepancies in numerical libraries," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 488–501.
- [59] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 785–796.
- [60] J. Liu et al., "Generation-based differential fuzzing for deep learning libraries," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 50:1–50:28, Dec. 2023. [Online]. Available: <https://doi.org/10.1145/3628159>.
- [61] Y. Liu et al., "Enhancing the interoperability between deep learning frameworks by model conversion," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1320–1330.
- [62] J. Bai et al., "ONNX: Open neural network exchange," 2019. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/onnx/onnx>
- [63] "bibatOss-fuzz." Google. 2021. Accessed: Sep. 9, 2022. [Online]. Available: <https://github.com/google/oss-fuzz>
- [64] C. Pacheco and M. D. Ernst, "RANDOOP: Feedback-directed random testing for Java," in *Proc. Companion 22nd ACM SIGPLAN Conf. Object-oriented Program. Syst. Appl. Companion*, 2007, pp. 815–816.
- [65] "Libfuzzer – A library for coverage-guided fuzz testing," LLVM. 2021. Accessed: Sep. 9, 2022. [Online]. Available: <http://llvm.org/docs/LibFuzzer.html>
- [66] X. Xie et al., "DeepHunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 146–157.
- [67] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Int. Conf. Mach. Learn.*, PMLR, 2019, pp. 4901–4911.
- [68] X. Zhang et al., "Predoo: Precision testing of deep learning operators," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2021, pp. 400–412.
- [69] 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.*, vol. 6, no. OOPSLA1, pp. 73:1–73:26, Apr. 2022. [Online]. Available: <https://doi.org/10.1145/3527317>.
- [70] W. Luo, D. Chai, X. Run, J. Wang, C. Fang, and Z. Chen, "Graph-based fuzz testing for deep learning inference engines," in *Proc. 43rd Int. Conf. Softw. Eng. (ICSE)*, IEEE Press, 2021, pp. 288–299, doi: 10.1109/ICSE43902.2021.00037.
- [71] J. Liu et al., "Finding deep-learning compilation bugs with NNSmith," 2022, *arXiv:2207.13066*, doi: 10.48550/arXiv.2207.13066.
- [72] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in tensorflow programs," in *Proc. 34th Eur. Conf. Object-Oriented Program. (ECOOP)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [73] D. Xiao, Z. LIU, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, Feb. 2022, doi: 10.1145/3508035.
- [74] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *J. Syst. Softw.*, vol. 84, no. 4, pp. 544–558, 2011.
- [75] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sep. 2016.
- [76] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *Proc. 2nd Int. Workshop Metamorphic Testing, (MET)*, Piscataway, NJ, USA: IEEE Press, 2017, pp. 28–34.
- [77] A. Dwarakanath et al., "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2018, pp. 118–128.
- [78] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 189–200.
- [79] J. Chen et al., "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, doi: 10.1145/3363562.
- [80] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, (PLDI)*, New York, NY, USA: ACM, 2014, pp. 216–226, doi: 10.1145/2594291.2594334.
- [81] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, New York, NY, USA: ACM, 2015, pp. 386–399, doi: 10.1145/2814270.2814319.
- [82] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, New York, NY, USA: ACM, 2016, pp. 849–863, doi: 10.1145/2983990.2984038.