SC4

An Automated Testing Tool for Deep Learning Libraries

by

Jingyu WU, Hongyuan CHANG, Jiachen ZHAO, Hong SU

SC4

Advised by

Prof. Shing Chi CHEUNG

Submitted in partial fulfillment of the requirements for COMP 4981

in the

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

2022-2023

Date of submission: April 20, 2023

Abstract

Artificial intelligence (AI) has ushered in a new era of technological advancements, but its reliance on deep learning libraries can introduce bugs that are difficult to detect and debug manually. This problem becomes more pronounced when the bugs are not immediately apparent, leading to costly and time-consuming efforts to identify and fix them. To address this challenge, we developed an automated testing tool for deep learning libraries that uses model-based techniques to detect bugs efficiently. Our experiments demonstrate that our tool successfully identified bugs in the libraries, validating its effectiveness and usefulness in ensuring the reliability of DL libraries.

CONTENTS

1 INTRODUCTION

1.1 OVERVIEW AND OBJECTIVES

1.1.1 Overview

The deep learning (DL) technique has become a powerful tool and has wide applications in many fields. Safety-critical applications in areas such as auto-driving and medical image recognition increase the importance of behavior related to correctness, robustness, privacy, and fairness. Unfortunately, like traditional software, DL applications can be shown to be vulnerable and lack robustness in various situations. Bugs not successfully detected before deployment could lead to unpredictable features and thus cause real-world devastating results, especially in safety-critical fields. For example, a recent report [17] has shown that nearly 400 driver-assistance technology-related auto-vehicle crashes happened in 2021, and [15] reports that a driver was killed while driving under the Tesla Autopilot mode. Thus, extensive and prevalent application of DL models in more and more safety-critical fields calls for extra attention to the testing of model validity and trustworthiness before deployment to real-world scenarios.

DL libraries are the framework and basis of building and deploying DL models. The correctness of DL systems and DL research is based on the correctness of DL libraries. That is to say, even with a feasible layer design and correct implementation, DL models can still go wrong with bugs contained in the underlying DL libraries. However, unlike bugs in traditional software, these bugs are even harder to detect. The unique complexity originating from the layer structure of the DL model and the elusive math implementation of the matrix calculations brings extra difficulty to the testing job. Packaged in high-level APIs, the bugs inside the libraries are more latent and hard to detect during production. Moreover, when a bug is detected during the model deployment stage, developers tend to examine the model design instead of testing the underlying

DL libraries, which further increases the difficulty of the whole testing process.

Testing has been demonstrated as an effective way to detect bugs while circumventing economic loss before massive deployment. With the recent rapid rise of interest, researchers have developed approaches to examine DL libraries. However, existing methods still cannot comprehensively exercise the DL libraries. The widely adopted API testing method working like a traditional unit test shows effectiveness in detecting bugs within each function of a DL library, but it is powerless when it comes to detecting bugs occurring at function interactions when the DL model stage changes. Besides, API-based methods usually read the constraints from the DL library documentation, and this could lead to low precision of testing. Model-generated methods, however, are able to test DL libraries more comprehensively through generating a bunch of random models, then loading the models with DL libraries, and finally detecting bugs through a series of predetermined metrics regarding the model training (or testing) results.

With a model-generated method, testing tools like Muffin [11] have successfully extended testing phases from execution to the training stage, covering low-level libraries, like TensorFlow [7], CNTK [1] and Theano [8], while PyTorch [6], one of the most popular DL frameworks, still remains uncovered. Besides, although the variation of models generated by Muffin has been breaking through in terms of the breadth of the libraries being successfully tested, there is still much room for improvement regarding the generated model structure diversity.

1.1.2 Objectives

1. Develop an automated testing tool that is compatible with PyTorch [6], ONNX [5], and TensorFlow [7].
2. Detect bugs in the model conversion and inference phase by first generating models in PyTorch and then converting the PyTorch models into other DL libraries for testing.

3. Detect discrepancies in documentation. For example, some constraints are required when using some APIs, which are not mentioned in the documentation.

4. Organize and upload our code to GitHub with clear documentation and a README file.

1.1.3 Challenges and Effort

1. **Extra engineering efforts were required, because we could not generate and run models in multiple DL libraries simultaneously.** Muffin [11] detects bugs in multiple DL libraries by utilizing multiple backends in Keras [3]. i.e., Keras can support multiple backends such as CNTK [1], TensorFlow [7], and Theano [8] simultaneously, which eases the coding job of Muffin. However, the latest version of Keras no longer supports multiple backends. Hence, we needed to first generate models in some DL libraries and use some conversion tools to convert them for future bug detection. As a result, the engineering effort was much larger, since we could not simply modify and extend the code of Muffin. We had to implement the tool almost from scratch instead.

2. **Less popularity of ONNX results in more human efforts to check the documentation or source codes.** ONNX [5] is relatively not that popular compared to other DL libraries such as PyTorch. Hence, there are much fewer tutorials and discussions about ONNX online. At the same time, the probability of discrepancies and mistakes in the ONNX documentation is higher. For example, there are almost no discussions about the gradient extraction API of an ONNX model except for its original documentation. However, the documentation of this extraction API is incomplete. For instance, there are typos in the documentation, and some parameters of the potential loss functions involved in this API are not supported. As a result, extra human effort was required to check the relevant documentation.

3. **The number of layers of models with the same structure in different DL libraries may not be the same, requiring more efforts to track the output layers.** The layer mapping

given the same model structure in different DL libraries may not always be one-to-one. For example, if we generated a PyTorch model with a Conv2D layer and converted it into an ONNX model, we found that this single Conv2D layer would map to tens of layers (subject to the ONNX version) in the ONNX model. As a result, it was challenging to find the ”output layer” (i.e., the layer generating the final output of Conv2D) in the ONNX model for differential testing. To tackle this problem, we proposed an algorithm to extract only the results of all ”output layers” of the models in other libraries. More details are provided in Section 2.2.3.

1.2 LITERATURE SURVEY

1.2.1 Deep Learning Framework Bugs

To build a detection system for identifying bugs in a deep learning framework, it is crucial to understand their characteristics. An extensive study was conducted by Chen, et al [10] to understand DL framework bugs by examining 800 bugs across four major DL frameworks, including TensorFlow [7], PyTorch [6], MXNet [4], and Deeplearning4j [2]. These bugs are categorized below and analyzed based on their root causes and symptoms.

Root Causes of Bugs

Among the total 13 root causes identified, the top-3 common root causes of bugs were incorrect algorithm implementation, type confusion and tensor shape misalignment.

- Incorrect Algorithm Implementation
  DL-related algorithms, such as computing gradients or performing operation fusion, are susceptible to this type of bug due to incorrect implementation logic. As DL research develops at a rapid pace, DL frameworks must be updated frequently to incorporate state-of-the-art (SOTA) algorithms. The rapid advancement in hardware also requires frameworks to provide implementations for new features. However, it is very likely that new bugs will be introduced during the implementation of new complicated code logic.

Thus, by extending algorithm coverage to include the latest implementations or testing in different hardware environments, more bugs may be detected.

- Type Confusion
  This kind of bug is caused by tensor operations, such as type checking or type conversion. In DL frameworks, tensors are multidimensional matrices composed of homogeneous type elements. Many bugs arise from the misuse of tensor type, either implicitly or explicitly, since almost all DL operations rely on tensors.

  As a result, adding type mutation operations to enrich the variety of input parameter types may help create suitable scenarios for triggering bugs. Obtaining valid and invalid input data types for a specific function is crucial for examining type checking and conversion functionalities.

- Tensor Shape Misalignment
  This kind of bug results from tensor shape-related problems, such as shape mismatches during inference and transformation. A tensor's shape indicates how many elements it contains in each dimension.

  According to this root cause, it may be possible to detect bugs at different stages of model execution by inserting layers with varying shapes during the model generation process. For example, generating extreme shapes such as tensors with size 0 may help detect bugs caused by incorrect edge case handling.

Bug Symptoms

There are six symptoms of DL framework bugs, with crashes and incorrect functionality being the most common.

- Crashes
  A DL framework crashes when it terminates unexpectedly for various reasons, such as running out of memory or getting a null pointer exception. There is an explicit oracle to detect crashes, and the error message often provides a hint for debugging. By designing

more informative and precise error messages, developers can identify the causes of the crashes more easily.

- Incorrect Functionality

  The framework performs problematically without crashing, such as having incorrect results, intermediate states, or model structures. It is more difficult to detect the underlying bugs related to this symptom, since there isn't an explicit oracle.

  Developers currently use oracle approximation (OA) with manually designed thresholds to detect incorrect functionality bugs in DL algorithms due to their stochastic nature [14]. Nevertheless, this poses maintenance challenges, since the threshold may need to be changed manually as the code evolves. It may be possible to alleviate this problem by creating a voting mechanism that uses multiple test metrics, since if one becomes invalid due to changes in code, the others could still remain valid.

1.2.2 Existing Testing Techniques

Since DL systems are designed to solve problems without known answers, testing them is challenging. In recent years, differential testing has been used primarily to overcome the oracle problem [9]. In the presence of multiple implementations of the same algorithm across different DL frameworks, differential testing detects bugs by checking whether different frameworks or different execution modes of the same framework produce different results for the same inputs and algorithms. Depending on the granularity of testing, these approaches can be further divided into API-based testing [19, 20] and model-based testing [11, 12, 16, 18]. API-based testing is similar to unit testing in that each function is tested separately, with input constraints provided by documents [20] or existing code [19]. However, these methods are limiting in detecting bugs generated during the interaction between different functions. In contrast, model-based testing involves executing a complete model, either from existing models [16], performing mutations on existing models [18] or generating models from some predefined structures [11]. Since model-based testing is performed on the entire model, abnormalities caused by function interaction can be detected. In the following subsections, three model-based approaches are introduced, CRADLE [16], LEMON [18] and Muffin [11].

CRADLE

CRADLE [16] was the first known attempt to detect and localize bugs in DL frameworks. It uses publicly available DL models as input to perform differential testing on three DL frameworks (TensorFlow [7], Theano [8], and CNTK [1]). Two metrics, D_CLASS and D_MAD, are proposed to distinguish between actual bugs and randomness caused by the stochastic nature of DL algorithms. D_CLASS is designed for classification models, and D_MAD can be used for both classification and regression models. CRADLE has been successfully used to detect 12 unique bugs in DL frameworks by evaluating 11 popular datasets and 30 existing models.

LEMON

Testing DL framework solely based on existing models has the obvious disadvantage of restricting the portion of code being tested. In order to maximize coverage of code being invoked during testing, LEMON [18] performs mutation operations on existing models, including mutations to layers, neurons, and weights. Additionally, LEMON utilizes a novel strategy that can guide the generation process toward producing models that can effectively amplify inconsistent signals for real bugs. This facilitates the process of distinguishing between real bugs and the uncertainty caused by randomness. As a result of these two improvements, LEMON has been able to detect 24 new bugs in four DL frameworks (TensorFlow [7], Theano [8], MXNet [4] and CNTK [1])

Muffin

Previous approaches (CRADLE [16] and LEMON [18]) primarily focus on detecting bugs in the model inference phase and do not examine code elsewhere in the DL framework. More specifically, previous methods have not adequately tested the logic for training DL models, which accounts for a significant portion of functionality coverage. Through a novel data trace analysis, Muffin [11] enables testing during the training phase. The model training process is divided into three stages, and a series of metrics are assigned for each stage to measure the amount of inconsistency.

Further, Muffin expands the model search space by utilizing a fuzzing-based approach for model generation instead of relying on existing models. Models are constructed layer by layer based on predefined structures, making them more diverse and achieving higher functionality coverage. Muffin has been used to successfully detect 39 newly discovered bugs by executing the latest release versions of three DL frameworks (TensorFlow [7], Theano [8], and CNTK [1]). However, Muffin is limited in the number of DL libraries that it can test. Therefore, in our project we extended its coverage to PyTorch[6] and ONNX[5].

2 METHODOLOGY

2.1 DESIGN

In this project, we developed an automated testing tool to perform testing on several DL libraries including PyTorch and TensorFlow. The core principles of our tool are:

1. Generate diverse DL models in PyTorch
2. Utilize the ONNX package to convert the PyTorch models into ONNX format
3. Execute models on PyTorch, ONNX Runtime and TensorFlow respectively
4. Detect potential bugs during the whole process, and log them into a database.

2.1.1 Model Generation

To generate diverse DL models, our tool utilizes the top-down generation algorithm proposed by Muffin [11]. The algorithm first generates model structure information, which determines the topology of the layer connection. Next, layer information for each layer is generated, which specifies the layer type and parameters. Then, our tool creates the corresponding PyTorch model given the generated information.

2.1.2 Model Conversion and Inference

After generating models in PyTorch, our tool converts them into ONNX format, which can be executed on ONNX Runtime as well as TensorFlow after further conversion into TensorFlow models. Next, it runs the converted models and performs inference on ONNX Runtime and TensorFlow respectively.

2.1.3 Testing

Finally, it detects bugs during these phases. Potential bugs may come from the following aspects: 1) documentation, which means the documentation has typos and provides insufficient or inconsistent information for users, 2) model conversion phase (e.g., when converting a model from library A to library B, because library B may not support some parameters in library A, the conversion may fail), and 3) model inference phase. Our tool utilizes differential testing methods to detect inconsistent bugs in phases 3). It logs all bugs, including crash, inconsistency, NaN, Inf bugs into a database for further analysis.

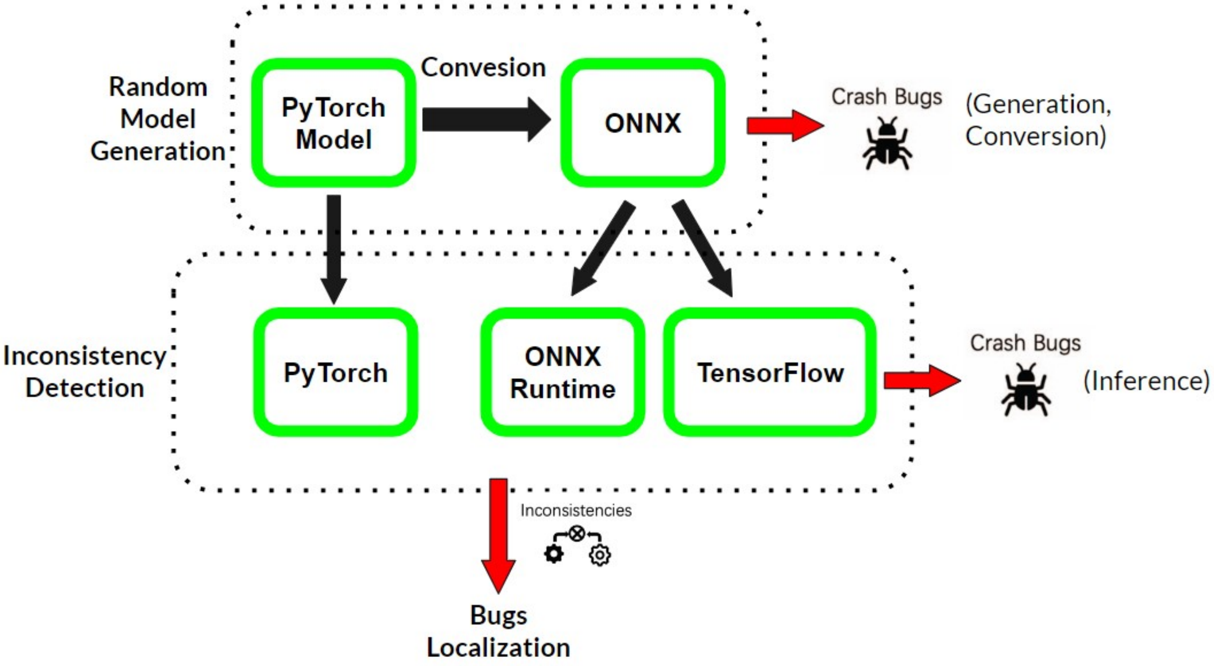


Figure 2.1. Overview of our tool.

2.2 IMPLEMENTATION

2.2.1 Model Generation

Structure Information Generation

The structure of DL models can be viewed as a directed acyclic graph (DAG), where each vertex represents a layer and each edge represents the corresponding link between layers. By randomly generating a DAG, out tool is able to construct diverse model structures.

Layer Information Generation

Given the model structure as a DAG, our tool needs to further decide the layer information of each vertex. We implemented a layer information generator which can randomly generate parameters for a specific layer type according to constrains specified in the documentation. Currently, more than 25 PyTorch layers are supported, including commonly used APIs such as Linear Layers, Convolutional Layers, MaxPool Layers, AvgPool Layers, BatchNorm Layers, Activation Layers, etc. For each vertex in the DAG, our tool generates a suitable layer information according to the input shape and parameter constrains, then calculates the corresponding output shape of the vertex.

Our tool adopts a non-comprehensive approach for specifying constraints of specific layer types. That is, for a parameter, our tool considers its constrains solely based on the parameter. As the constraints of a parameter may depend on the value of another parameter, this approach cannot guarantee the generation of a valid layer. Therefore, we implemented a error handler that are able to detect invalid generated layers and perform re-generation until a valid layer is generated. This approach strikes a good balance between the engineering efforts required to embed comprehensive constraints into the layer generation process and the proportion of valid layers generated while constructing a model, which affects the efficiency of our tool.

In order to generate models with diverse layer types, we adopted a layer selection process that assigns layer types that have been rarely used a higher chance of being selected. For any layer type *L*, we record the number of times that *L* has been selected to construct a model, denoted as *t*. A score

$$\mathrm{s} = \frac{1}{t+1} \tag{2.1}$$

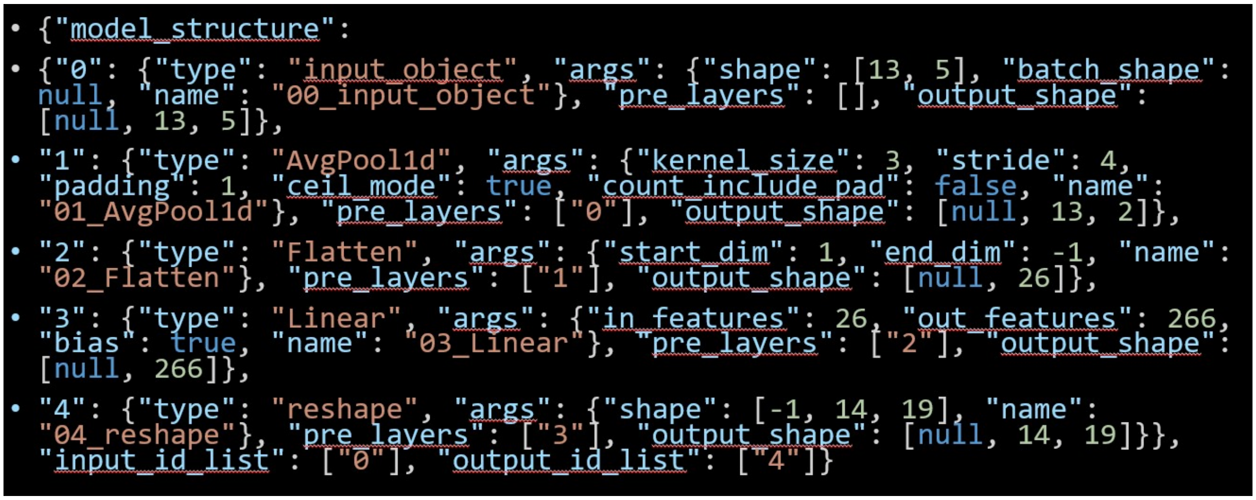
will be assigned to *L*, the probability that layer type *L* will be selected would be

$$\mathrm{p} = \frac{\mathrm{s}}{\sum_{k=1}^{r} s_k} \tag{2.2}$$

, where r is the total number of possible layer types.

The generated model structure and layer information is stored in a dictionary, as shown in Fig. 2.2. The corresponding model is shown in Fig. 2.3.

In addition, We have implemented a python module call TorchModel, which can take the dictionary storing the generated structure and layer information as input to create a PyTorch nn.Module object. Therefore, we can utilize the TorchModel class to convert the structure and layer information into an actual PyTorch model. The TorchModel module is designed to support any DAG model structure as input. During initialization, corresponding torch.nn layer objects are created according to the layer information. Afterward, each layer is connected according to the structural information by traversing the input DAG.



```
• {"model_structure":
• {"0": {"type": "input_object", "args": {"shape": [13, 5], "batch_shape":
  null, "name": "00_input_object"}, "pre_layers": [], "output_shape":
  [null, 13, 5]},
• "1": {"type": "AvgPool1d", "args": {"kernel_size": 3, "stride": 4,
  "padding": 1, "ceil_mode": true, "count_include_pad": false, "name":
  "01_AvgPool1d"}, "pre_layers": ["0"], "output_shape": [null, 13, 2]},
• "2": {"type": "Flatten", "args": {"start_dim": 1, "end_dim": -1, "name":
  "02_Flatten"}, "pre_layers": ["1"], "output_shape": [null, 26]},
• "3": {"type": "Linear", "args": {"in_features": 26, "out_features": 266,
  "bias": true, "name": "03_Linear"}, "pre_layers": ["2"], "output_shape":
  [null, 266]},
• "4": {"type": "reshape", "args": {"shape": [-1, 14, 19], "name":
  "04_reshape"}, "pre_layers": ["3"], "output_shape": [null, 14, 19]}},
  "input_id_list": ["0"], "output_id_list": ["4"]}
```

Figure 2.2. Generated model structure and layer information

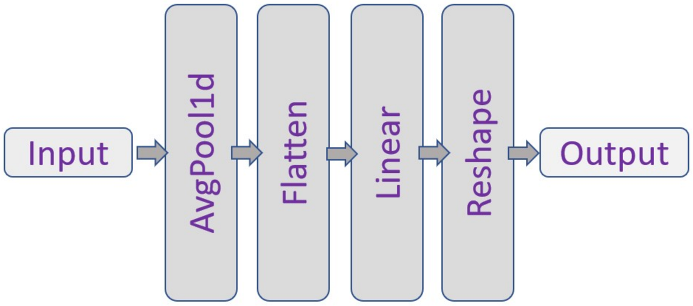


Figure 2.3. Generated PyTorch model

2.2.2 Model Conversion

After generating models in PyTorch, we needed to convert the PyTorch models into other DL libraries to enable differential testing. We mainly utilized the package ONNX [5] to convert the PyTorch models into ONNX format. The major API is the function torch.onnx.export(), which saves the structure and weights of a PyTorch model in a file with the suffix .onnx.

2.2.3 Testing

In the following sections, we mainly introduce our implementation to detect various bugs in different phases.

Documentation Discrepancy Detection

Bugs may also exist in the documentation of these DL libraries. The bugs here mainly refer to the insufficient/inconsistent/wrong information provided by the documentation. For example, when using the *export_gradient_graph* API to export gradients from an ONNX model, we found that one parameter *loss_fn* has incomplete documentation. For instance, the documentation claims that *CrossEntropyLoss()* will work for this API. However, we found that ONNX does not support probability inputs for *CrossEntropyLoss()* here, which is not stated in the documentation.

Conversion Bugs

Conversion bugs refer to bugs that emerged in the DL model conversion phase. For instance, a typical type of conversion bug is the parameter-not-supported error. For example, some parameters are only supported in PyTorch but not supported in ONNX. What is more, due to the possibility of documentation discrepancies, as in Section 2.2.3, such parameter-not-supported information may not be stated in the documentation. As a result, it is inconvenient for users to use these DL libraries, especially for those who are interested in model conversion.

The way we analyzed the conversion bugs was that we logged every error in the conversion phase into a database, and then we analyzed them manually later.

Crash Bugs and NaN and Inf Bugs

Crash bugs in DL models are bugs that make the program crash when doing inference. These bugs usually happen because of certain combinations of layers. To determine where the bugs come from, we needed to infer from the model structure. Hence, if such a crash happened, our tool reported the bug and saved the model information, especially the model structure, in our database for further analysis.

NaN and Inf bugs are another type of bug that gives NaN or Inf values in the model output. Special inputs together with the wrong implementation of the functions trigger this kind of bug. This kind of bug is easy to localize, since the first layer that gives the incorrect output is the buggy layer. When such a bug was detected, our tool reported the bug and saved the model information, inputs and the mediate outputs in the database.

Inconsistent Bugs

Inconsistent bugs emerge in scenarios such that given the same input and the same configurations, two models in different DL libraries produce different outputs. For instance, if given the same input and the same weights, two models in different DL libraries give different inference output, it indicates that there are some bugs in the forward calculation phase in one of them.

In our project, we tested inconsistent bugs in the model inference phase. Differential testing is applied here to detect the inconsistencies. The metric we used is the distance in the L^p space [13]. Formally, given two vectors $\mathbf{x}, \mathbf{y}$ in the $L^p(p \in \mathbb{N}^+ \bigcup \{+\infty\})$ space with the number of basis elements as $\mathcal{N}$, the L^p distance is defined as follows

$$D(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_p = \left(\sum_{i=1}^{\mathcal{N}} |x_i - y_i|^p \right)^{1/p} \tag{2.3}$$

If the L^p distance of the output tensors in different DL libraries is significant given the same configurations, we suggest the existence of inconsistent bugs. In this case, we store the model information, inputs and the intermediate outputs in the database for localization.

Output Layer Mapping

As mentioned at the 3-rd point in Section 1.1.3, one layer in a model of a library A (e.g., PyTorch library) may correspond to multiple layers in another library B (e.g., ONNX library) after model conversion. As a result, it is critical to only extract the intermediate outputs of the "output layers" such that we can perform the differential testing to localize the buggy layers.

It is worth noticing that only extracting the outputs of "output layers" is not a trivial task particularly in terms of the computational cost or manual efforts. Our primary solution is to explicitly extract the names of all the possible output layers for models converted into other DL libraries. For example, given a set of layers (e.g., Conv2D, MaxPooling) that are used to generated PyTorch models, we use each of the layer to generate a PyTorch model and convert it into ONNX or TensorFlow models. Intuitively, the last layer of the converted layer corresponds to the "output layer". If we could store all these tuples *(PyTorch_layer_name*, *ONNX_layer_name*, *TF_layer_name)*, we can successfully tackle the challenge.

However, the primary solution has two major limitations. 1) The mapping results are "static", which means every time we increase the size of the PyTorch layer set, we need to update the mapping "output layer" sets. 2) The extraction procedure of intermediate results of the converted model is also inconvenient. For example, we need to use the name of the layer to extract the output of the “output layer”. However, if a model contain two or more layers with the same type, extracting their outputs by name may cause chaos (e.g., by indexing problem).

To remedy these limitations, we improve our algorithm of extracting outputs of the “output layers” in other DL libraries. We first make a reasonable assumption that for a l-th layer in a model, the existence of all the $(l+k)$-th $(k \in \{1, 2, 3, \cdots\})$ will not affect the output of the l-th layer. As a result, we proposed the following algorithm as Algorithm 1. Formally, given a PyTorch model with $\mathcal{K}$ layers, the goal of Algorithm 1 is to return two lists of outputs, ONNX “output layer” list O_1 and TensorFlow “output layer” list O_2. The length of O_1 and O_2 are both $\mathcal{K}$.

Algorithm 1

```
Given: A set of layers of the target PyTorch model $\{l_1, \cdots, l_{\mathcal{K}}\}$
       An input of the target model $\mathcal{I}$
$\mathcal{L} \leftarrow \{\}$                                        ▷ A temp layer set
$O_1 \leftarrow \{\}$
$O_2 \leftarrow \{\}$
for $l$ in $\{l_1, \cdots, l_{\mathcal{K}}\}$ do
    $\mathcal{L} \leftarrow \mathcal{L} \cup l$
    Use $\mathcal{L}$ to generate a PyTorch model $\mathcal{P}$
    Convert $\mathcal{P}$ into ONNX model $\mathcal{O}$ and TensorFlow model $\mathcal{T}$
    $o_1 \leftarrow$ get prediction of $\mathcal{O}$ given $\mathcal{I}$           ▷ Compute ONNX output
    $o_2 \leftarrow$ get prediction of $\mathcal{T}$ given $\mathcal{I}$     ▷ Compute TensorFlow output
    $O_1 \leftarrow O_1 \cup o_1$
    $O_2 \leftarrow O_2 \cup o_2$
end for
return $O_1, O_2$
```

Notice that we only run this mapping algorithm after we detect a significant difference in the model outputs of different DL libraries to avoid adding too much overheads.

Differential Testing

An effective way to detect inconsistent bugs is by conducting differential testing of intermediate outputs, such as forward computation and loss computation [11, 12]. For example, Muffin [11] uses a way to detect inconsistencies in forward computation as follows.

$$Inc\_Fc = \big\{ l_i, i \in [1, n] \,\big|\, \big(D(O^i_j, O^i_k) > t\big) \wedge \big(D(O^p_j, O^p_k) < \epsilon, p \in P(i)\big)\big\} \tag{2.4}$$

where Inc_Fc is the set of all inconsistent layers with respect to forward computation, n denotes the total number of layers in the generated model, l_i denotes the i-th layer, O_j^i and O_k^i denote the outputs of l_i using library j and k, and $P(i)$ denotes the set of all the predecessors of l_i. t and ϵ are two user-defined thresholds to detect how significant should the outputs in different libraries to be considered inconsistent.

As suggested by Equation 2.4, the forward computation inconsistencies will only be discovered if the output of a certain layer differs significantly but with insignificant variation in the outputs for previous layers.

Localization of Buggy Layers

There are two major types of bugs, 1) inconsistent bugs and 2) crash bugs or NaN (Not a Number) bugs. And we propose two approaches to test the their localization respectively.

For inconsistent bugs, the localization method is following the idea in Section 2.2.3 and Equation 2.4. i.e., We assume the inconsistencies are caused by the first inconsistent layer [11]. After we localize the buggy layers, to avoid false positives, we plan to further check the buggy layers manually by checking the corresponding source codes and so forth.

For the remaining bugs including crash bugs, NaN bugs, and Inf bugs, since most of the code editors or compilers will keep track of where the error/bugs occur, we may mainly learn the localization of buggy layers by reading the console outputs.

2.3 TESTING

2.3.1 Testing the Effectiveness on Buggy Layer Localization

As mentioned in Section 2.2.3, due to the fact that error may accumulate, simply setting some threshold and choosing the first significant inconsistent layer may lead to the wrong localization of inconsistent buggy layers. Hence, we propose two approaches to remedy it.

Correspondingly, we propose some methods to test the effectiveness of our ideas. 1) The first approach is to run the same model with different thresholds with a wide range. To test whether we can use this strategy to find the threshold with the most suitable sensitivity, we can focus on the difference in the number of inconsistencies given different thresholds. For instance, we try to avoid choosing extreme threshold values that result in either too many inconsistencies or almost no inconsistencies. 2) To test whether our scores assignment algorithm is effective, we can manually design some layers that must generate wrong outputs. For example, we can manually implement a wrong Conv1D layer and insert it into some models. Then run our algorithm. If this wrong layer gets a top-k score, then our algorithm is proven to be effective.

2.3.2 Testing the Effectiveness of Crash Bugs Detection

For crash bugs, we need to ensure the crash is caused by the DL library itself but not our implementation to ensure the effectiveness of our detection. We propose several ways to validate our effectiveness as follows. 1) For crash bugs in the model generation and model conversion phases, we will repeatedly access and test the same APIs to ensure it is not an occasional crash. 2) For crash bugs in the inference/backward phases, we only count the crash bugs when at least one of the DL libraries can function properly to ensure the crash is not caused by our tool.

2.4 EVALUATION

We have done experiments with our tool. The results show that our tool is fully functional. We now evaluate our system based on the objectives stated in Section 1.1.2.

1. We have implemented an automatic testing tool that is compatible with PyTorch, ONNX and TensorFlow. Our tool can support model conversion and model inference testing.

2. We have generated $N = 1500$ testing samples. We then analyze the portion of model generation failure and crashes as shown in Figure 2.4. As we can see, a large portion of model conversions fail. Some of the possible reasons may be unsupported parameters in different DL libraries or insufficient constraints. For example, we manually inspected

some of the generation failures and found that some of them were caused by layer conversion that is unsupported by ONNX, and some other errors were either due to insufficient constraints during the model generation phase or inappropriate conversion settings. Detailed error messages can be found in Appendix D.

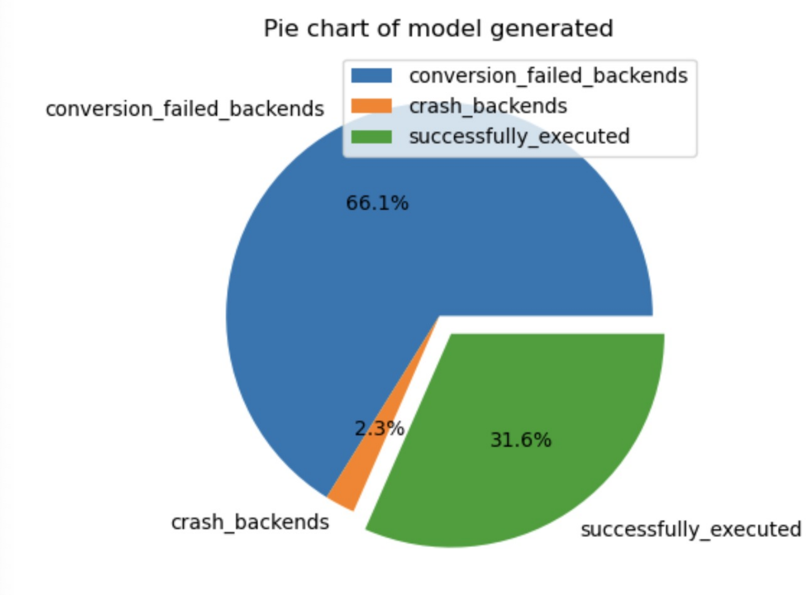


Figure 2.4. Among 1500 testing samples, a high portion of model conversions fail due to various reason such as the unsupported parameter issue or crash bugs.

3. We continually record potential improvements for documentation of popular DL libraries. Some of our records can be found in Appendix D.

4. The source code of the project can be found in https://github.com/Hong-YC/fyp_dl_library_testing.git. The code is well commented. We will add a README in the future.

3 DISCUSSION

3.1 THRESHOLD DETERMINATION

In our project, determining the appropriate threshold for the model_output_delta variable is a critical task. If the threshold is set too low, it could detect a large number of inconsistencies that are unrelated to bugs, whereas setting it too high could result in a low bug-detection rate. To find the optimal threshold value, we conducted experiments and created diagrams to visualize the number of inconsistencies detected with varying threshold values.

As depicted in Figure 3.1, increasing the threshold value results in a significant reduction in the number of inconsistencies detected across different backends, particularly at lower threshold values. The dropping trend becomes less pronounced as the threshold value increases.
We observed that the number of inconsistencies detected remains relatively stable when the threshold value exceeds 0.0001. Based on these results, we select 1e-4 as our threshold to achieve reliable detection of inconsistencies, while minimizing the number of false positives.

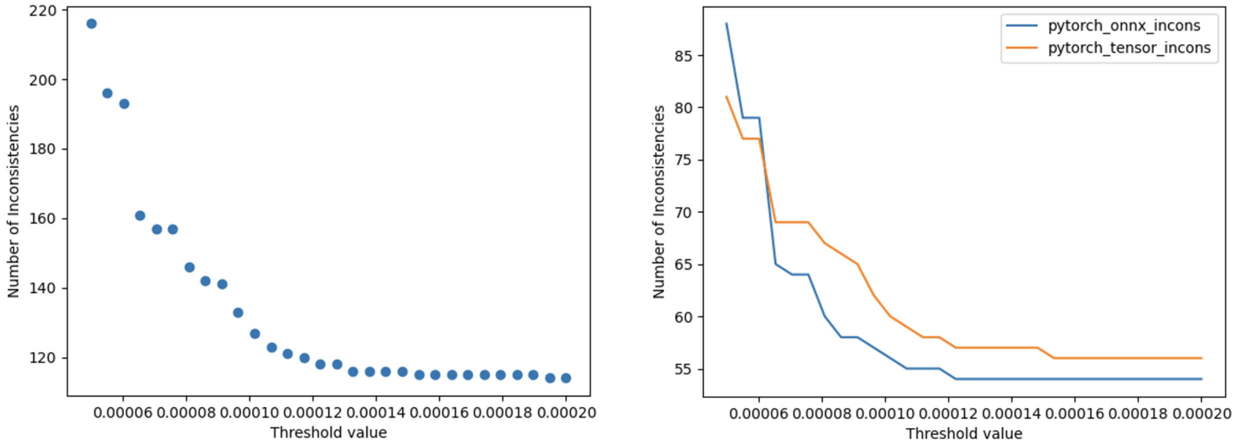


Figure 3.1. The left figure shows the total number of inconsistencies detected over different thresholds, and the right figure shows the number of inconsistencies detected in different backends.

Layer	BatchNorm1d	BatchNorm2d	BatchNorm3d	MaxPool3d	Conv2d
Num. of Inconsistencies	28	11	6	4	4
Fraction	0.56	0.52	0.3	0.17	0.15

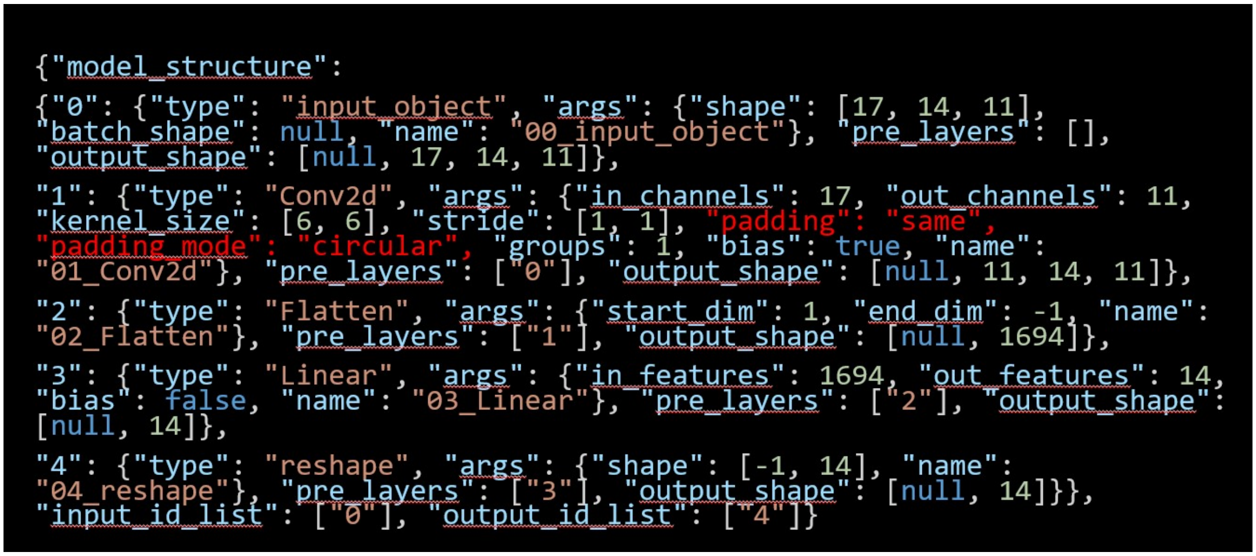
Table 3.1. We select the top-5 layers with the highest fraction of inconsistent fraction over $N = 1500$ testing samples.

3.2 INCONSISTENT LAYERS ANALYSIS

After selecting the threshold for differential testing, we generating $N = 1500$ models for testing. The next goal for us is analyzing which layers are more suspicious to contain inconsistent bugs. To do so, we count the number of inconsistencies occurred on each layers. What is more, we also compute the fraction of inconsistent counts over the total number of appearance for each layer. For instance, for each layer type (e.g., Conv2d), if it occurs to be inconsistent for k times, and it was generated K times over the N models, we compute the fraction as $\frac{k}{K}$. We demonstrate our analysis in Table 3.1. As we can see, BatchNorm1d layer type has the highest inconsistent fraction, which suggests that BatchNorm1d is suspicious to contain inconsistent bugs. As a result, we further analyze this layer type in the next sub-section.

3.3 BUGS DETECTED

In our experiments, our tool was able to detect an inconsistency bug related to the ”Conv1d” and ”Conv2d” layers. The structure of the model in which the bug was detected is shown in Fig. 3.2. We observed significantly different outputs when we ran the PyTorch and ONNX versions of this model. To investigate this issue further, we conducted additional research. Interestingly, we found this bug is only triggered with the following parameter setting: padding = ”same”, padding_mode = ”circular”. As Convolution layers are widely used layers, this is quite an important discovery. At the same time, due to the popularity of PyTorch, we conjecture that the error comes from the ONNX conversion or inference implementation. We have taken the initiative to report this issue on the ONNX forum and are hopeful that the developers will be able to confirm and address this bug.



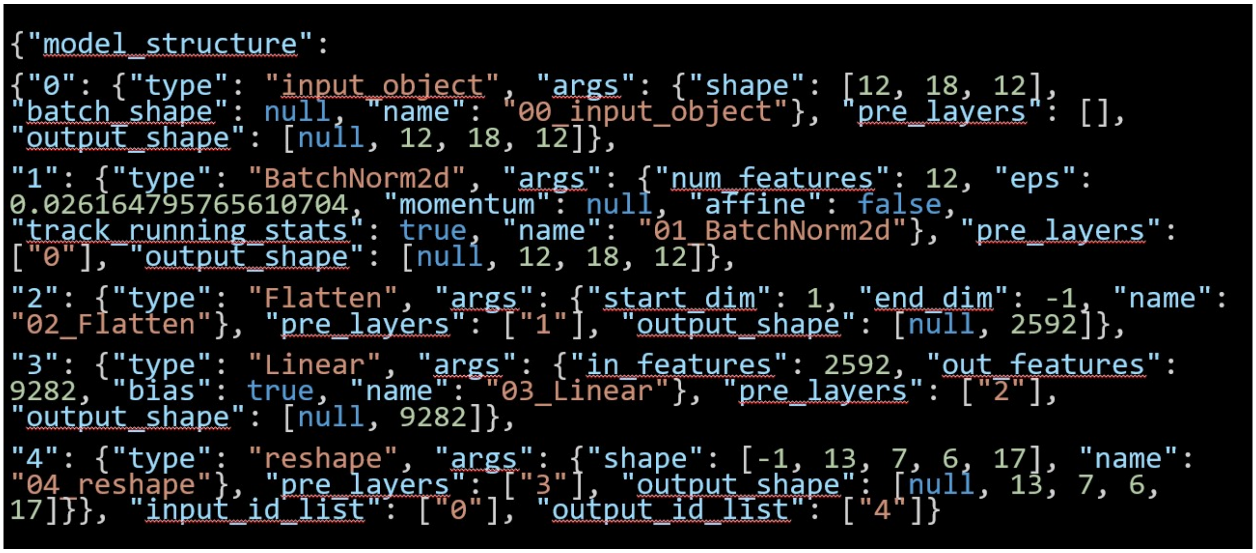
```
{"model_structure":
{"0": {"type": "input_object", "args": {"shape": [17, 14, 11],
"batch_shape": null, "name": "00_input_object"}, "pre_layers": [],
"output_shape": [null, 17, 14, 11]},
"1": {"type": "Conv2d", "args": {"in_channels": 17, "out_channels": 11,
"kernel_size": [6, 6], "stride": [1, 1], "padding": "same",
"padding_mode": "circular", "groups": 1, "bias": true, "name":
"01_Conv2d"}, "pre_layers": ["0"], "output_shape": [null, 11, 14, 11]},
"2": {"type": "Flatten", "args": {"start_dim": 1, "end_dim": -1, "name":
"02_Flatten"}, "pre_layers": ["1"], "output_shape": [null, 1694]},
"3": {"type": "Linear", "args": {"in_features": 1694, "out_features": 14,
"bias": false, "name": "03_Linear"}, "pre_layers": ["2"], "output_shape":
[null, 14]},
"4": {"type": "reshape", "args": {"shape": [-1, 14], "name":
"04_reshape"}, "pre_layers": ["3"], "output_shape": [null, 14]}},
"input_id_list": ["0"], "output_id_list": ["4"]}
```

Figure 3.2. The structure of the model with an inconsistency bug. The unique layer is ”Conv2d”.

Another bug our tool detected is related to the “BatchNorm2d” layer. The structure of the model in which the bug was detected is shown in Figure 3.3. We observed significantly different outputs when we ran the PyTorch and ONNX versions of this model. To investigate this issue further, we conducted additional research. Interestingly, we discovered that the inconsistency bug only occurred when we performed the conversion to ONNX before running the inference on the PyTorch model. In contrast, running the inference prior to the conversion yielded nearly identical results. We also observed similar behaviors in the layers “BatchNorm1d” and “BatchNorm3d”. We suspect the problem comes from the conversion process from the PyTorch model to the ONNX format. We have taken the initiative to report this issue on the ONNX forum and are hopeful that the developers will be able to confirm and address this bug.

3.4 BUGS UNDETECTED

In our experiments, we have not encountered any occurrences of NaN or Inf bugs within the deep learning libraries. However, these findings do not necessarily imply an absence of such bugs in the libraries. The lack of identified issues may be attributed to a limited testing scope, or potentially



```
{"model_structure":
{"0": {"type": "input_object", "args": {"shape": [12, 18, 12],
"batch_shape": null, "name": "00_input_object"}, "pre_layers": [],
"output_shape": [null, 12, 18, 12]},
"1": {"type": "BatchNorm2d", "args": {"num_features": 12, "eps":
0.026164795765610704, "momentum": null, "affine": false,
"track_running_stats": true, "name": "01_BatchNorm2d"}, "pre_layers":
["0"], "output_shape": [null, 12, 18, 12]},
"2": {"type": "Flatten", "args": {"start_dim": 1, "end_dim": -1, "name":
"02_Flatten"}, "pre_layers": ["1"], "output_shape": [null, 2592]},
"3": {"type": "Linear", "args": {"in_features": 2592, "out_features":
9282, "bias": true, "name": "03_Linear"}, "pre_layers": ["2"],
"output_shape": [null, 9282]},
"4": {"type": "reshape", "args": {"shape": [-1, 13, 7, 6, 17], "name":
"04_reshape"}, "pre_layers": ["3"], "output_shape": [null, 13, 7, 6,
17]}}, "input_id_list": ["0"], "output_id_list": ["4"]}
```

Figure 3.3. The structure of the model with an inconsistency bug. The unique layer is "BatchNorm3d".

due to not discovering specific input combinations that would trigger the corresponding bugs.

3.5 UNSOLVED PROBLEMS

In our original project objectives, we aimed to incorporate a training phase testing feature in our tool. However, we faced significant engineering challenges that prevented us from achieving this goal. Despite using the DL library ONNX to convert models into different libraries, we discovered that the existing APIs in ONNX were insufficient to produce the desired results. One major issue we encountered was that the TensorFlow model, when converted from an ONNX model, was incapable of being trained. Instead, it functioned more like an ONNX model that used TensorFlow to perform inference, rather than a genuine TensorFlow model.

Although we had several ideas for addressing this obstacle, we were unable to put them into action. One such idea was to limit the number of data in the training set to just one to reduce randomness during the training phase. This approach could serve as a promising direction for future research and development.

4 CONCLUSIONS

4.1 SUMMARY

Throughout our project, we conducted extensive research on previous work, with a particular focus on Muffin[11]. By studying the workflow of Muffin[11], we were able to build upon its foundation and develop our own methodology. In the implementation stage, we achieved successful model generation and conversion, which were crucial for the feasibility of our project. By utilizing models with identical structures and weights from different libraries, we were able to conduct inference and perform differential testing, ultimately resulting in the discovery of bugs in the libraries. Our findings demonstrate the effectiveness of our tool in detecting bugs and its applicability in real DL library development.

4.2 RECOMMENDATIONS FOR FUTURE WORK

Our tool can be further extended to expand the testing range. There are two possible directions: 1) Add the training phase testing to the tool. 2) Apply our method to other libraries.

Testing training phase Our tool only tests the bugs in the inference phase. Another important phase in the DL libraries is the training phase, involving losses and gradients calculation. We didn't include this part because there are no proper APIs in the libraries for us to extract the losses and gradients. If the developers of the ONNX library release new APIs, it's possible to implement testing for the training phase. It would significantly expand the testing range.

Extension to other libraries Our tool tests three DL libraries, PyTorch, ONNX Runtime, and TensorFlow. Some other DL libraries, e.g. Caffe2, are also widely used. Our method can be applied to other libraries as long as the ONNX supports conversion to those libraries. It would not only help users and developers working with those libraries but also help increase the accuracy of current testing results since our method is based on comparisons among models of different libraries.

5. REFERENCE

[1] Cntk. https://docs.microsoft.com/en-us/cognitive-toolkit/, Accessed: 2023.

[2] Deeplearning4j. https://deeplearning4j.konduit.ai/, Accessed: 2023.

[3] Keras. https://keras.io/,Accessed:2023.

[4] Mxnet. https://mxnet.apache.org/versions/1.9.1/, Accessed: 2023.

[5] Onnx. https://github.com/onnx/onnx, Accessed:2023.

[6] Pytorch. https://pytorch.org/, Accessed: 2023.

[7] Tensorflow. https://www.tensorflow.org/?hl=zh-tw, Accessed: 2023.

[8] Theano. https://theano-pymc.readthedocs.io/en/latest/, Accessed: 2023.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[10] J. Chen, Y. Liang, Q. Shen, and J. Jiang. Toward understanding deep learning framework bugs, 2022.

[11] J. Gu, X. Luo, Y. Zhou, and X. Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing, 2022.

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

[13] I. J. Maddox. *Elements of functional analysis*. CUP Archive, 1988.

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

[15] J. F. Peltz and S. Masunaga. Fatal crash of tesla model s in autopilot mode leads to investigation by federal officials, Jun 2016.

[16] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038, 2019.

[17] D. Saul. Nearly 400 crashes in past year involved driver-assistance technology-most from tesla, Jun 2022.

[18] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 788–799, New York, NY, USA, 2020. Association for Computing Machinery.

[19] A. Wei, Y. Deng, C. Yang, and L. Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source, 2022.

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

APPENDIX

A REQUIRED HARDWARE & SOFTWARE

A.1 HARDWARE REQUIREMENTS

- GPU: Tesla M60, TITAN V
- CPU: Intel(R) Core(TM) i7-8700K @ 3.70GHz

A.2 SOFTWARE REQUIREMENTS

- Pytorch[6]
- Tensorflow[7]
- ONNX[5]

B PROJECT PLANNING

Work	7	8	9	10	11	12	1	2	3	4	5
Do the literature survey	▨	▨	▨								
Reproduce result of existing methods		▨									
Build the tool			▨	▨	▨	▨					
-Analyze source code for Muffin			▨								
-Research on PyTorch, ONNX and TensorFlow			▨	▨							
-Model generation				▨	▨	🟧					
-Structure generation modules				▨	▨						
-Layer information modules				▨	▨	🟧					
-Inference testing modules					▨	▨	🟧				
-Implementing pipeline					▨	▨	🟧				
-Training phase testing modules						░	▨	🟧			
Run our tool								░	▨	▨	
Design more diverse model generation structure								░	▨	▨	
Design new metrics for detecting inconsistency								░	▨	▨	
Test our implementation								░	▨	▨	
Write the proposal report			▨								
Write the monthly reports				▨	▨	▨	▨				
Write the progress report							▨	▨			
Write the final report										▨	
Prepare for the presentation										▨	
Presentation video										▨	

Work Distrubution

▨ Work Distribution

Work	Jingyu WU	Hongyuan CHANG	Jiachen ZHAO	Hong SU
Do the literature survey	▨			
Reproduce result of existing methods		▨		
Extend Muffin to Pytorch			▨	
-Analyze source code for Muffin				▨
-Research on Pytorch	▨			
-Implement Pytorch version of different functionality modules of Muffin		▨		
-Model generation			▨	
-Structure genertion modules				▨
-Layer information modules	▨			
-Inconsistency detection module		▨		
-Implement training workflow			▨	
-Integrate Pytorch implementation into Muffin				▨
Design more diverse model generation structure	▨			
Design new metrics for detecting inconsistency		▨		
Test our implementation			▨	
Write the proposal report				▨
Write the monthly reports	▨			
Write the progress report		▨		
Write the final report			▨	
Prepare for the presentation				▨
Presentation video	▨			

C MEETING MINUTES

The appendix should be formatted the same as the chapters.

C.1 MINUTES OF MEETING ON 0708

Date: July 8, 2022

Time: 08:00 PM

Place: Zoom

Present: Ziniu, Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jingyu

1. **Approval of minutes**

   This was the first formal group meeting, so there were no minutes to approve.

2. **Report on Progress**

   All team members have started reading the research papers which were posted on Google drive.

3. **Discussion items**

   - Ziniu has introduced the group to two cutting-edge methods of DL framework: Muffin and GraphFuzz. He also gave suggestions on group work planning and task division.

   - Hongyuan has raised some questions about the papers read last week.

4. **Goals for the coming week**

- All members will go read the aforementioned two papers (GraphFuzz, Muffin) and the source code.
- Contact cssystem@cse.ust.hk to apply for a server for fyp usage (GPU: RTX2080Ti).
- Draft task schedule for the following year, divide work for paper reproduction. source: https://course.cse.ust.hk/comp4471/UG_GPU.pdf

5. **Meeting Adjournment and Next Meeting** The meeting was adjourned at 9:15 PM. The next meeting will be in approximately two weeks, but hasn't been set yet. The place will be set later by e-mail.

C.2 MINUTES OF MEETING ON 0720

Date: July 20, 2022

Time: 08:00 PM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Hong

1. **Report on Progress**

   Continue reading paper

2. **Discussion items**

   About meeting with Ziniu. This summer we intend to meet two times, one is after we determine the outline of the proposal. The other one is after we finish the proposal.

3. **Goals for the coming week**

   - **Read Papers** Finish reading all the given related papers. Finish reading all model-based papers and have the. Read CRADLE first, and MUFFIN is the state-of-the-

art model.
After reading all API-based papers, have another meeting to decide whether to choose API-based or model-based approach.

- **Get Familiar with UGCPU Provided by HKUST**
  source: https://course.cse.ust.hk/comp4471/UG_GPU.pdf

4. **Meeting Adjournment and Next Meeting** The meeting was adjourned at 8:45 PM. Next meeting on 2022 July 27 8 p.m

C.3 MINUTES OF MEETING ON 0729

Date: July 29, 2022

Time: 10:00 PM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Hongyuan

1. **Report on Progress**
   Finish reading Model-based paper

2. **Discussion items**

   - Model-based testing -¿ how to create a model, layer
   - Inconsistency -¿ any difference (how to calculate?)
   - location threshold -¿ how to set? (Same with others, justified reasons, metric)
   - Model base cons: amplify the error?
   - Does any function have an internal stage?

- Can we improve the rate of generating valid models?

Ziniu: Muffin uses Keras for model building/inference/training, it is difficult to explore the input space for middle level API and low-level API -¿ directly use low-level API to build model

Possibility -¿ code to generate model using low-level API (pytorch, tensorflow etc..)

Low-level -¿ much higher manual efforts (function input constraints)

Use API-based mining tools to obtain constraints -¿ then apply to model-based

3. **Goals for the coming week**

   - **Read Papers** Read API-based paper

4. **Meeting Adjournment and Next Meeting**

   The meeting was adjourned at 11:00 PM.

   Next meeting on 2022 Aug 5 9 p.m

C.4 MINUTES OF MEETING ON 0805

Date: August 05, 2022

Time: 09:00 PM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jiachen

1. **Report on Progress**

   Continue reading papers

2. **Discussion items**

- **Discuss and summarize the papers**

  Discuss different approaches of API-based fuzzing test in papers

- **Discuss possible direction**

  API-based:

  Obtain constraint directly from source code (mainly computation bugs)

  Model-based:

  Use API constraint to test model

  Use text mining to obtain input and output format. Use the format to generate model

3. **Goals for the coming week**

   - Try running Muffin on server and reproducing the result
   - Read code if possible

4. **Meeting Adjournment**

   The meeting was adjourned at 9:55 PM.

C.5 MINUTES OF MEETING ON 0902

Date: September 02, 2022

Time: 10:30 AM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Hong

1. **Plan**

   First implement Pytorch (top priority). To do this, first read the source code of Muffin.

Later depends on our progress and time remaining, we will finish some of those following tasks,

- Extend to support Pytorch
- Different model generation structure
- Improve inconsistency detection metrics
- Find out why some generated model will lead to unsupported-crashes, and how can we avoid it
- Run on some new datasets

2. **Planning structure of Proposal**

   - Explain the real-world context for our project
   - Define the problem to address and briefly discuss existing solutions and their limitations
   - show the niche in our project
   - Define your deliverables and success criteria (extends to PyTorch and discover some bugs in PyTorch)
   - Outline your project schedule

3. **Deadline for writing introduction Methodology**

   Sep 6

4. **Proposal Distribution**

Overview & Objectives	Wu Jingyu
Literature survey	Chang Hongyuan
Design & Implementation	Zhao Jiachen
Testing & Evaluation	Su Hong

C.6 MINUTES OF MEETING ON 0906

Date: Sep 06, 2022

Time: 10:00 PM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Hongyuan

1. **Report on Progress**

   Draft session “Introduction”, “Literature Review” and “Methodology” for proposal report.

2. **Discussion items**

   Discuss GANTT chart

   - Do literature survey (7 - 9)
   - Reproduce result of existing methods (8)
   - Analyze source code for Muffin (9)
   - Research on Pytorch (9 - 10) Implement Pytorch version of different functionality modules of Muffin (10-12)
     - Structure genertion modules (10-11)
     - Layer information modules (10-11)
     - Inconsistency detection module (10-11)
     - Implement training workflow (10 -11)
     - Integrate Pytorch implementation into Muffin (12)
   - Design more diverse model generation structure (11-3)

- Design new metrics for detecting inconsistency (11-3)
- Test our implementation (1-3)
- Write the proposal report (9)
- Write the monthly reports (10, 11, 1)
- Write the progress report (1-2)
- Write the final report (4)
- Prepare for the presentation (4)
- Presentation video (4)

3. **Goals for the coming week**

- Finalize Proposal (GANTT chart, work distribution)
- Start reading Muffin source code
- Schedule meeting with Ziniu

4. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:50 PM
Depend on availability of Ziniu (Before 9.12)

C.7 MINUTES OF MEETING ON 0909

Date: Sep 09, 2022
Time: 8:30 PM
Place: Zoom
Present: Ziniu, Hong, Jingyu, Hongyuan, Jiachen
Absent: None

Recorder: Hongyuan

1. **Report on Progress**

   Finish draft of proposal report

2. **Discussion items**

   Ziniu gives us some useful advice after reading our report.

   - Introduction: Add more content: challenges, significance of testing model (add motivating examples), description of model based method
   - ONNX model conversion Test pytorch: Muffine TensorFlow/Keras, ONNX, PyTorch
   - Why listing common root cause? what is the difference from normal software, how can we obtain better design by knowing the root causes
   - Metric improvement: Discuss the limitation of current metric, draw data from our experiment
   - Test: Add localization, (finish module then unit test)
   - Evaluation: how diverse is our model compare to existing methods is also a performance improvement
   - General: Change structure to architecture (for example include layer parameters(Like Audee))
   - Add hardware: Cpu, Gpu
   - Add software: Pytorch, tensorflow, ONNX

3. **Goals for the coming week**

   - Modify proposal report

- Start reading code of Muffin

4. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:15 PM

Next meeting: 9.23 8:30 p.m.

C.8 MINUTES OF MEETING ON 0926

Date: Sep 26, 2022

Time: 9:00 AM

Place: Zoom

Present: Professor Cheung, Ziniu, Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jingyu

1. **Discussion items**

Advice from Prof and Ziniu:

- if the team choose to continue using Keras to construct the models -¿ hard to outperform. (Better generate more diversified models not using Keras, but to invoke low-level API of the DL library; Currently, low-level API bugs can not be detected due to the limitation of the models generated by Keras)
- Comments on page 14: Apply the inconsistency check logic in the same library (different libraries may not have the same API calls) For metamorphic testing for DL library testing, you may refer to this paper: EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries
- How to measure the diversity of the model generated? Metric regarding

model diversity needs to be considered (percentage of API coverage? -Jiachen) (multiple diversity metric: layer diversity, structure diversity? -Prof)

- How to localize the bug? Where the crash appears is not necessarily where the collision happens
- Think about how to extend Muffin in the bug detection phase (will be easier compared to extending model complexity)(bug in optimizer?)

2. **Goals for the coming week**

- Explore the above-mentioned directions, try to propose relative solutions
- Continue reading code of Muffin

3. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:15 PM
Next meeting: 9.23 8:30 p.m.

C.9 MINUTES OF MEETING ON 1121

Date: Nov 21, 2022
Time: 9:00 AM
Place: Zoom
Present: Professor Cheung, Ziniu, Hong, Jingyu, Hongyuan, Jiachen
Absent: None
Recorder: Jingyu

1. **Report on Progress**

In this month, our team has successfully built an elementary pipeline for pytorch library bug detection including: chain-based model generation, conversion into ONNX runtime model, and comparison of the consistency of the forward computation stage.

2. **Discussion items**

Professor and Ziniu have pointed out the following problems:

- Chain-based models we have achieved generation now are too simple for bug-detection.
- Avoid relying on Pytorch for all the model validation works, try to add the validation in the model generation stage. Only leave the complex case to Pytorch for checking.
- Try to generate all possible models, and try to reduce the number of invalid model generated at the same time
- Try to utilize the onnx runtime for training, or explore some other modules besides inference for testing.

3. **Goals for the coming week**

- Improve the current pipeline according to the aforementioned direction.
- Integrate the current jupyter notebook pipeline into a python file.

4. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:15 AM

Next meeting: 12.5 9:00pm

C.10 MINUTES OF MEETING ON 0119

Date: Jan 19, 2023

Time: 9:00 PM

Place: Zoom

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jingyu

1. **Discussion items**

- Jiachen: One layer of the Pytorch model is converted into several layers in the ONNX model (problem, How to match ONNX layers with Pytorch layers? Linear models are handleable, use dictionary to record mapping.)
- Hong: Did literature review of gradient extraction of ONNX. Few resources findable online.

2. **Goals for the coming week**

- Hongyuan: Write a function in experiment.py to generate a model with one single layer (every currently generated layer); Continue pipeline construction.
- Hong: Continue research on gradient extraction
- Jingyu: Continue generating layers
- Jiachen: Resolving the aforementioned problem. Keep experimenting.

3. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:00 PM

Next meeting: 1.23

C.11 MINUTES OF MEETING ON 0201

Date: Feb 01, 2022

Time: 9:00 PM

Place: Zoom

Present: Ziniu, Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jingyu

1. **Report on Progress**

   Model generation pipeline finished. Inference testing done, loss calculation testing done, onnx training done (batch size set to 1, see if the randomness can be eliminated)

2. **Discussion items**

   Ziniu gives us some useful advice on the following direction.

   - Hong: Difficulty in extracting gradients from the onnx model transformed from the Pytorch model.
   - Ziniu proposed link: Gradient — onnxcustom
   - Jiachen: only support onnx with version newer than 12th
   - Ziniu: Theoretically feasible to extract gradient from onnx, may read source code. (too heavy engineering workload)
   - may test the gradient calculation of pytorch later

3. **Goals for the coming week**

   - jiachen: finish loss calculation testing
   - Jingyu: layers to be added: activation layers, pooling layers, RNN, normalization,
   - Ziniu: coverage criteria of the diversity and inclusiveness of model generation [2208.01508] COMET: Coverage-guided Model Generation For Deep Learning Library Testing (Jingyu, Hongyuan)
   - Hongyuan: add function to detect invalid generated layer and throw away.

possible metrics of our work, next stage work after 2.14:

- code coverage (one person learn
- how many bugs we detect
- the diversity of our models (see section 3.2 of paper https://arxiv.org/abs/2208.01508)(most difficult) one person learn
- at least 20 layers to be added before the report is submitted next week.
- add layer configuration constraint (https://github.com/maybeLee/COMET/blob/master/implementations/boostraps/api_implementations/api_config_pool.json)
- next stage priority: testing ¿¿ localization (find bug first, then localization)

4. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 10:15 PM

Next meeting: 9.23 8:30 p.m.

C.12 MINUTES OF MEETING ON 0315

Date: March 15, 2023

Time: 8:00 PM

Place: Campus

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jiachen

1. **Discussion items**

- Hong: Proposed a new method to solve the problem of extracting the intermediate outputs of the TensorFlow model.
- Jiachen: Followed Hong's idea, suggested that applying the same method on ONNX and PyTorch.
- Hongyuan and Jingyu: Report their own works.

2. **Goals for the coming week**

- Hongyuan: Continue completing the pipeline.
- Hong: Implement the proposed method.
- Jingyu: Continue generating layers
- Jiachen: Help Hong with the work.

3. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 9:00 PM

C.13 MINUTES OF MEETING ON 0413

Date: April 13, 2023

Time: 5:00 PM

Place: Campus

Present: Hong, Jingyu, Hongyuan, Jiachen

Absent: None

Recorder: Jiachen

1. **Discussion items**

Reported progress, discussed the final report, made appointment with the language tutor.

2. **Goals for the coming week**

    - Hongyuan: Continue completing the pipeline.
    - Hong and Jingyu: Run experiments.
    - Jiachen: Start the report.

3. **Meeting Adjournment and Next Meeting**

The meeting was adjourned at 6:30 PM

D EXPERIMENT RESULT

D.1 ERROR RECORD

- torch.onnx.errors.UnsupportedOperatorError: Exporting the operator ’aten::_convolution_mode’ to ONNX opset version 12 is not supported. Please feel free to request support or submit a pull request on PyTorch GitHub: https://github.com/pytorch/pytorch/issues

- torch.onnx.errors.UnsupportedOperatorError: Exporting the operator ’aten::fractional_max_pool2d’ to ONNX opset version 12 is not supported. Please feel free to request support or submit a pull request on PyTorch GitHub: https://github.com/pytorch/pytorch/issues

- torch.onnx.errors.UnsupportedOperatorError: Exporting the operator ’aten::fractional_max_pool3d’ to ONNX opset version 12 is not supported. Please feel free to request support or submit a pull request on PyTorch GitHub: https://github.com/pytorch/pytorch/issues

- torch.onnx.errors.OnnxExporterError: Unsupported: ONNX export of operator avg_pool2d, divisor_override. Please feel free to request support or submit a pull request on PyTorch GitHub: https://github.com/pytorch/p

- RuntimeError: The serialized model is larger than the 2GiB limit imposed by the protobuf library. Therefore the output file must be a file path, so that the ONNX external data can be written to the same directory. Please specify the output file name.

- ValueError: Attempted to use an uninitialized parameter in <method ’detach’ of ’torch._C._TensorBase’ objects>. This error happens when `you are using a ‘LazyModule‘ or explicitly manipulating ‘torch.nn.parameter.Unini objects. When using LazyModules Call ‘forward‘ with a dummy batch to initialize the parameters before calling torch functions`

- `RuntimeError: mat1 and mat2 shapes cannot be multiplied (1x48 and 240x8)`

D.2 ERROR IN DOCUMENTATION

- onnxruntime.training.experimental.gradient_graph._gradient_graph_tools.export_gradient_graph. For the documentation in http://www.xavierdupre.fr/app/onnxcustom/helpsphinx/api/onnxruntime_python/grad.html, the Python version documentation regarding *loss_fn* has typos. The following *Web* and *instead* bullet points should also be included into *loss_fn*. At the same time, the documentation states that *CrossEntropyLoss()* is supported. But the probability input format is not supported for *CrossEntropyLoss()* in this API.