The Journal of Systems & Software 201 (2023) 111681

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Deep learning framework testing via hierarchical and heuristic model generation ☆

Yinglong Zou, Haofeng Sun, Chunrong Fang *, Jiawei Liu, Zhenping Zhang

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, China
Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai, 201112, China

ARTICLE INFO

Article history:
Received 6 December 2022
Received in revised form 13 February 2023
Accepted 20 March 2023
Available online 24 March 2023

Keywords:
Software testing
Deep learning framework
Hierarchical and heuristic model generation
Precision bug

ABSTRACT

Deep learning frameworks are the foundation of deep learning model construction and inference. Many testing methods using deep learning models as test inputs are proposed to ensure the quality of deep learning frameworks. However, there are still critical challenges in model generation, model instantiation, and result analysis. To bridge the gap, we propose Ramos, a hierarchical heuristic deep learning framework testing method. To generate diversified models, we design a novel hierarchical structure to represent the building block of the model. Based on this structure, new models are generated by the mutation method. To trigger more precision bugs in deep learning frameworks, we design a heuristic method to increase the error triggered by models and guide the subsequent model generation. To reduce false positives, we propose an API mapping rule between different frameworks to aid model instantiation. Further, we design different test oracles for crashes and precision bugs respectively. We conduct experiments under three widely-used frameworks (TensorFlow, PyTorch, and MindSpore) to evaluate the effectiveness of Ramos. The results show that Ramos can effectively generate diversified models and detect more deep learning framework bugs, including crashes and precision bugs, with fewer false positives. Additionally, 14 of 15 are confirmed by developers.

1. Introduction

In recent years, with the rapid development of deep learning (DL) techniques, many tasks can be well performed, such as computer vision (Voulodimos et al., 2018) and natural language processing (Chowdhary, 2020). DL techniques are widely used in various safe-critical domains in practice, such as automatic driving (Rao and Frtunikj, 2018) and disease diagnosis (Ju et al., 2019; Ji et al., 2019). DL models are critical components of DL techniques. Developers use DL frameworks to build and deploy DL models. In DL frameworks, algorithms for calculating tensors are implemented as API calls named DL operators. In other words, DL frameworks are the foundations of the DL model construction and inference. Therefore, the quality of DL frameworks is significant for that of DL models.

However, some empirical studies (Zhang et al., 2018) show that DL frameworks could raise bugs in DL models. Many testing methods are proposed to ensure the quality of DL frameworks. Among them, some methods (Zhang et al., 2021b) test every single operator. They call each API provided by the framework one by one and have a high interface coverage. Unfortunately, they fail to consider the operator interaction and thus miss some bugs, e.g., the channel inconsistency between two operators. To trigger bugs in DL frameworks more comprehensively, researchers propose some methods using DL models as testing inputs to test DL frameworks (Gu et al., 2022). These methods generally include three stages. In the **model generation** stage, new models are generated as test inputs. In the **model instantiation** stage, the newly generated models are instantiated into code. In the **result analysis** stage, the newly generated codes are executed, and the execution results are compared. These methods all adopt differential testing and use relative errors as test oracles. They instantiate and execute models under different frameworks. Then they calculate the differences in execution results of the same model under different frameworks as relative errors. A precision bug is triggered if relative errors exceed the given threshold. However, there are still challenges in these three stages.

Model Generation. Some researchers (Pham et al., 2019; Wei et al., 2022) use publicly available DL models (e.g., GCN (Zhao et al., 2019; Chen et al., 2019), rtm3d (Li et al., 2020)) as testing inputs. These methods can be restricted because it is impractical to collect a large number of public models that can invoke different portions of frameworks. As a solution, many researchers

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail addresses: yinglongtt@outlook.com (Y. Zou), SaitamaHF@outlook.com (H. Sun), fangchunrong@nju.edu.cn (C. Fang), jw.liu@smail.nju.edu.cn (J. Liu), snoozingcat@163.com (Z. Zhang).

https://doi.org/10.1016/j.jss.2023.111681
0164-1212/

design methods to generate new models under two requirements. Firstly, they aim to generate models with high diversity. Generally, models with higher diversity invoke more portions of frameworks, which contributes to triggering more bugs. Secondly, they expect to generate models with errors larger than the threshold. An error larger than the threshold implies a precision bug.

There are two state-of-the-art methods in DL framework testing, Muffin (Gu et al., 2022) and LEMON (Wang et al., 2020b). Muffin designs a layer selection procedure to give a larger chance to the operator that is rarely used before and records the error of each model. LEMON designs many model mutation rules to generate new models by changing existing models and proposes a heuristic strategy to guide model generation toward amplifying relative errors.

However, in the model variety, Muffin only increases the variety of operators but ignores the variety of model structures. Specifically, Muffin only modifies the operator type of given templates but does not modify the structure of the templates. LEMON only changes few operators of existing models, making the newly generated models similar to existing models. All of the above will lead to the decline of model diversity and thus miss some bugs.

Besides, in the model error, Muffin fails to feedback on model errors to the model generation. Because the mutation amplitude of models in LEMON is small, the heuristic strategy of LEMON does not achieve satisfactory results. In Section 5, we generate models by Muffin and LEMON respectively. The results show that few models generated can trigger errors larger than the threshold. To obtain higher-diversity models, we design a new hierarchical structure that supports modifying the templates. In this way, the hierarchical structure contributes to increasing the amplitude of model mutation. To obtain the model with larger errors, we take model errors as heuristic indicators to guide the structure with larger errors as the seed of the next round of mutation.

Model instantiation. Existing methods usually (Wang et al., 2020b; Gu et al., 2022) instantiate models by calling Keras, a high-level framework. To perform differential tests, they invoke different low-level frameworks (e.g., TensorFlow and PyTorch, see Section 2.2) just by choosing different backends for Keras without directly calling any APIs in low-level frameworks. Unfortunately, there are differences in parameter settings under different frameworks. When setting parameters for Keras, only the common parameters of all frameworks are given, and the different parameters between frameworks are set as the default values by Keras. Therefore, there may be inconsistent results of the same model under different frameworks, which does not meet the preconditions of differential testing. So some bugs detected are false positives. To solve this problem, we analyze the implementation of each framework API and sort out the differences in parameter settings among different frameworks. Based on these differences, we propose an API mapping rule among frameworks and apply it to the model instantiation.

Result Analysis. In addition to crashes, precision bugs are critical because large-scale numerical operations are often carried out in DL applications. Many methods set a threshold for the error to detect precision bugs. However, it remains a problem to determine an appropriate value for the threshold. Most methods (Wei et al., 2022) provide thresholds casually. To make the testing oracle for precision bugs more convincing, we provide an error threshold by investigating the precision tolerance of the built-in test methods of each framework to detect precision bugs.

To sum up, we propose Ramos, a hierarchical and heuristic DL framework testing method. Firstly, Ramos obtains models with a mutation-based hierarchical model generation method, which uses a novel hierarchical structure to increase the amplitude of mutation. In the hierarchical structure, both the operators in the templates and the structures of the templates can be modified to improve the diversity of models. Ramos adopts an error-oriented heuristic method to increase the error of models. Secondly, when instantiating models, Ramos applies a mapping rule between different frameworks to deal with the differences in parameter settings. At last, Ramos provides an error threshold by investigating the precision tolerance under different frameworks to detect precision bugs. To evaluate Ramos, we conduct experiments under three widely-used frameworks (TensorFlow, PyTorch, and MindSpore). We compare Ramos against three state-of-the-art methods (LEMON (Wang et al., 2020b), Muffin (Gu et al., 2022), and Cradle (Pham et al., 2019)). The experimental results show that Ramos can generate models with higher diversity. The error triggered by Ramos is at least an order of magnitude larger than the other methods. And thus Ramos can trigger more precision bugs. Ramos successfully detects 15 crashes and 154 precision bugs, and 14 of these crashes are confirmed. The main contributions of this paper are as follows:

- We propose Ramos, a hierarchical heuristic DL framework testing method. In Ramos, we design a hierarchical structure to generate models with higher diversity and propose an error-oriented heuristic method to generate the model in the direction of larger error.
- We summarize the API mapping rule between different frameworks to avoid false positives.
- We implement the tool and design experiments to verify the proposed Ramos. Experimental results show that Ramos is excellent in the detection of precision bugs.

More information is available on the online package: https://github.com/zyltt/JSS2022.

2. Background

2.1. Deep learning model

DL models are structures in which layers are stacked to perform specific tasks (e.g., computer vision (Voulodimos et al., 2018) and natural language processing (Chowdhary, 2020)). Each layer corresponds to a function call that converts the input and weight to the output. A DL model can be mapped to a DAG (directed acyclic graph). The DAG is composed of a vertex set $V_G = \{v_1, v_2, \ldots\}$ and an edge set $E_G = \{e_1, e_2, \ldots\}$. Each vertex in V_G represents a tensor, while each edge in E_G represents a function call corresponding to a layer in the DL model. Suppose there is a directed edge e_k from node v_i to node v_j, and e_k represents the function call corresponding to layer $layer_k$. In that case, it means that the tensor represented by v_i is the input of $layer_k$, and the output tensor is passed to v_j after computation. The branches of the DAG are regarded as the branches of models.

Many artificially designed DL models are composed of repeated units. For example, ResNet (He et al., 2015) is composed of multiple residual units, and GoogLeNet (Szegedy et al., 2014) is composed of multiple inception units. Thus, the NAS (neural architecture search) algorithm is inspired to search the whole neural structure by searching the unit (cell) instead of the whole neural structure. NAS algorithm searches for two types of cells, including normal cells and reduction cells. The output and input dimensions of the normal cell are consistent, and the output size of the reduction cell is half the input size. After searching the cells, the two kinds of cells are stacked in a preset way to construct a complete model. In this paper, in order to make the models closer to reality, we refer to the model generation method of NAS. Fig. 1 shows how cells are stacked in models generated by Ramos. During the stacking process, the reduction cell repeats stacking $\lambda - 1$ times, followed by a normal cell. The size of λ is manually set in the configuration. Similar to HNAS, to moderate the model size, λ in Ramos is 3.

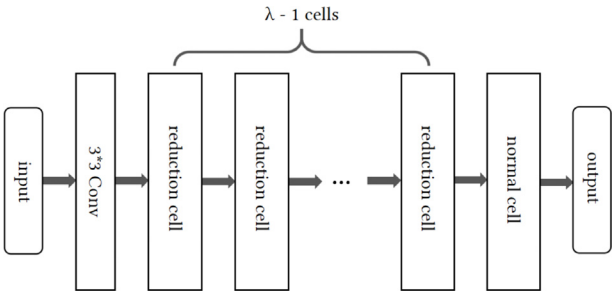

Fig. 1. Structure of DL model in Ramos.

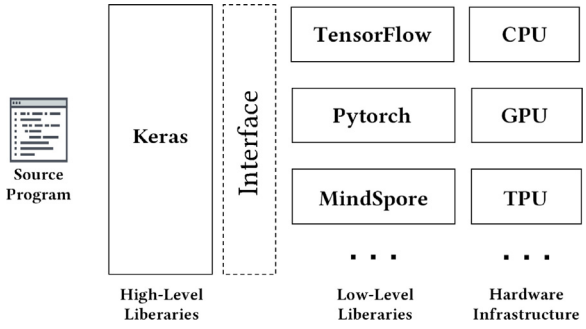

Fig. 2. Structure of DL framework.

2.2. Deep learning framework

DL framework is the cornerstone of DL model construction and inference. DL framework provides APIs for DL developers. Each API corresponds to an operator (Shatnawi et al., 2018). Fig. 2 shows the structure of the DL framework. Generally, the API in the high-level DL framework (e.g., Keras) calls the corresponding API in the low-level frameworks. Besides, the API in low-level frameworks can also be called by the source program directly. Similar to the existing works (Gu et al., 2022; Wang et al., 2020b; Guo et al., 2020), we focus on testing three widely-used low-level frameworks, including TensorFlow (Google, 2015; Abadi et al., 2016), PyTorch (Meta, 2016; Paszke et al., 2019), and MindSpore (Huawei, 2019).

The parameter setting between APIs under different DL frameworks is various. For example, when calling *Convolution*, we need to specify the size and initializer of the convolution kernel under MindSpore, while we can directly set the convolution kernel by passing a numpy array under PyTorch. These differences should be noted.

3. Hierarchical structure

In state-of-the-art methods (e.g., Muffin (Gu et al., 2022), LEMON (Wang et al., 2020b), and Cradle (Pham et al., 2019)), only one or few layers of a model are modified in one mutation. These methods are difficult to replace operator combinations in the model, resulting in low model diversity. In order to increase the diversity of the model, we take the following measures to replace the operator combinations in the model. Firstly, we design a hierarchical structure to represent operator combinations in the model. Then, we propose three heuristic indicators to measure the contribution to model error. Based on the proposed hierarchical structure and heuristic indicator, we design a novel mutation to modify the operator combinations in the model. In Section 3.1, we introduce the proposed hierarchical structure in detail. The heuristic indicators and mutation for the hierarchical structure are introduced in Sections 3.2 and 3.3.

3.1. Introduction of hierarchical structure

The schematic diagram of a hierarchical structure is shown in Fig. 3. A hierarchical structure contains L layers, each containing one or several motifs. Layer L contains only one motif. A motif is a directed acyclic-connected graph with a single source and single sink. In the hierarchical structure, each motif represents an operator combination, and it is composed of three main ingredients: **Available operator and operator combination set** $O = \{o1, o2, \ldots\}$. **Vertex set** $V = \{v_1, v_2, \ldots\}$. Each vertex in V corresponds to a tensor. **Edge set** $E = \{e_1, e_2, \ldots\}$. Each edge in E represents an operator or operator combination in O. If an edge e_i, which represents an operator or operator combination o_k, points from a vertex v_m to another vertex v_n, it means that the tensor corresponding to v_m is the input of the operator or operator combination o_k, and after computation, the output tensor is saved to v_n. If there are multiple edges (e.g., e_j, e_k) pointing to the vector v_n, these output tensors will be fused into one tensor (e.g., *Concatenate* on the channel dimension) and then saved to v_n.

We define the operator combination corresponding to the mth motif in the lth layer as $o_l^{(m)}$ and the available operator and operator combination set O of this motif as $O_l^{(m)}$. Then $O_l^{(m)} = O_{pri} \cup O_{l-1}$, where O_{pri} is the set of all the operators and $O_{l-1} = \left\{o_{l-1}^{(1)}, o_{l-1}^{(2)}, \ldots, o_{l-1}^{(n_{l-1})}\right\}$, n_{l-1} is the number of motifs in layer $l-1$. That is to say, O_{l-1} is a set of all operator combinations represented by motifs in layer $l-1$. It is worth noting that when $l = 1$, $O_1^{(m)} = O_{pri}$.

The hierarchical structure is used to generate a new model. A new model can be generated by the following steps. Firstly, Ramos mutates an existing hierarchical structure to generate a new hierarchical structure. Secondly, Ramos conducts a normal cell and a reduction cell from each newly generated hierarchical structure. Thirdly, as introduced in Section 2.1, a model is constructed by stacking $\lambda - 1$ reduction cells and a normal cell. Therefore, Ramos generates a new model by cloning the constructed reduction cell $\lambda - 2$ times and then stacking these $\lambda - 1$ reduction cells with the constructed normal cell.

3.2. Heuristic indicators for hierarchical structure

We design three heuristic indicators to measure the contribution to model error. Specifically, we design *fitness* to record the contribution of each hierarchical structure to model error. We propose *basic_weight* and *composite_weight* to record the contribution of each operator and operator combination to model error, respectively. *Fitness* is used in Section 4.3 to guide the selection of the hierarchical structure. It determines the probability that a hierarchical structure of the corpus will be selected as the seed. *Basic_weight* and *composite_weight* are used in Sections 3.3 and 4.3 to guide the selection of the operator and operator combination. *Basic_weight* and *composite_weight* respectively determine the probability that an operator or an operator combination represented by a motif will be selected in the heuristic mutation. In initialization (Section 4.2), we set three same default values for the *fitness* of each hierarchical structure, the *basic_weight* of each operator, and the *composite_weight* of each motif. During feedback (Section 4.6), the value of these indicators will be modified according to the model's results.

3.3. Mutation for hierarchical structure

State-of-the-art methods usually generate new models by mutation. Cradle (Pham et al., 2019) and LEMON (Wang et al., 2020b) design mutation rules (e.g., adding a layer, deleting a layer) and

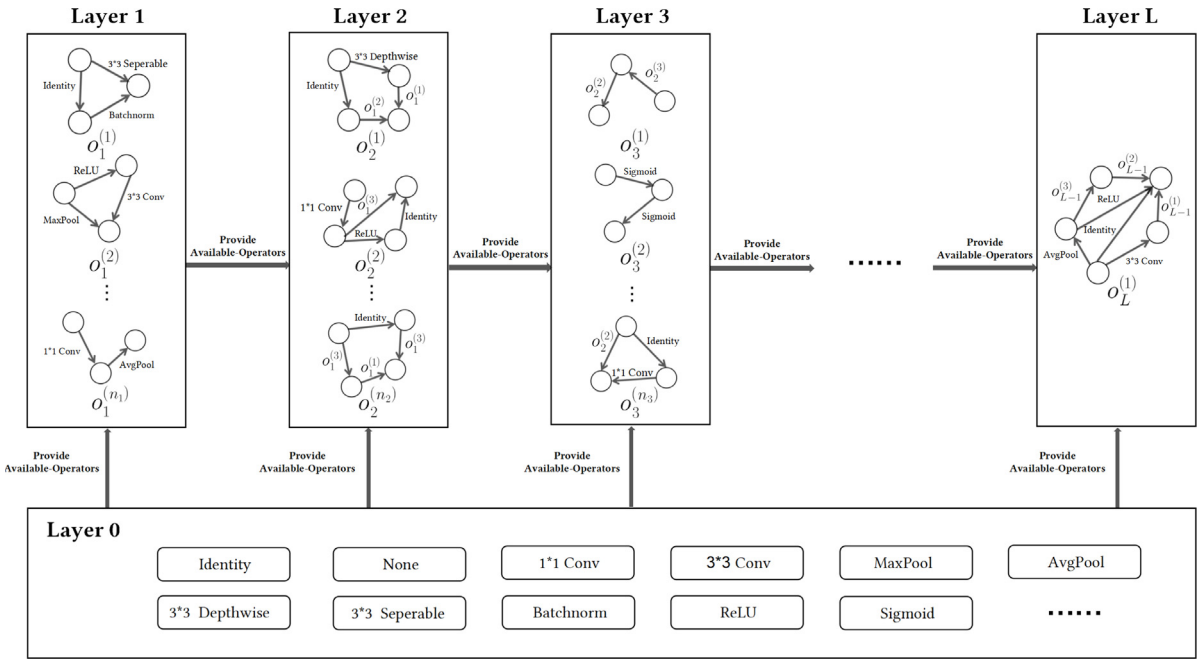

Fig. 3. Hierarchical structure.

modify the model directly. Because there are few mutation rules, this mutation has little modification to the model. Muffin (Gu et al., 2022) regards the model as a directed graph. Each operator corresponds to an edge in the graph. Muffin generates new models by determining each edge in the graph. However, the edge in this graph cannot represent an operator combination. Therefore, this method cannot replace the operator combination in the model. We design a novel mutation based on the proposed hierarchical structure. In the mutation, we regard each hierarchical structure as a directed graph and mutate it by replacing the edge in the graph. Since the edge in the graph can represent both operator and operator combination, our mutation can modify both operator and operator combination in the hierarchical structure. Because the new model is generated based on the mutated hierarchical structure, the modification of the operator combination is shown in the newly generated model.

We design two mutation modes for the hierarchical structure, including random mutation and heuristic mutation. The random mutation is used in initialization without the outputs of models. The heuristic mutation is used based on the outputs of models generated before.

The random mutation for a hierarchical structure is as follows:

(1) Randomly sample a target layer from the hierarchical structure.

(2) Randomly sample a target motif in the sampled target layer.

(3) Randomly sample a predecessor vertex i (i is not the sink) in the sampled motif.

(4) Randomly sample a successor vertex j ($j > i$) in the sampled motif.

(5) Replace the operator or operator combination from i to j with a randomly sampled one from the available set O. The probability of choosing an operator or an operator combination is t and $1-t$ respectively.

(6) Check whether the motif is valid. If not, jump to Step (1). Firstly, it is a directed acyclic connected graph, which is the basic requirement of the model construction. Secondly, there are only one source and one sink in the motif. Otherwise, there may be multiple inputs or outputs for models.

In Step (5), if an operator is replaced by the operator *None*, it means deleting the operator. If the operator *None* is replaced by another operator, it means inserting an operator. In this way, Ramos supports template modifications, which the other methods cannot achieve. Therefore, the mutation for the hierarchical structure increases the diversity of models.

The heuristic mutation process is similar to the random mutation process. In Step 5, the type of operator or operator combination is chosen randomly in random mutation. However, in heuristic mutation, Ramos calculates the probability as follows. While choosing an operator, the probability of each operator to be sampled is:

$$p = \frac{basic\_weight}{\sum_{i=1}^{r} basic\_weight_i}$$

where r is the number of operators. $Basic\_weight$ is a heuristic indicator introduced before.

While choosing an operator combination, the probability of each operator combination to be sampled is:

$$P = \frac{composite\_weight}{\sum_{i=1}^{n_l} composite\_weight_i}$$

where n_l is the number of motifs contained by layer l. $Composite\_weight$ is a heuristic indicator introduced before.

In heuristic mutation, when an operator or combination of operators is sampled, if the error of the newly generated model increases, the probability that the operator or combination of operators will be selected again will also increase, so that models with larger and larger errors can be generated.

It is worth noting that the mutation based on the hierarchical structure is more effective than that in state-of-the-art methods (including Muffin, LEMON, and Cradle). On the one hand, our method can support all the modification rules (e.g., adding a layer, deleting a layer) in state-of-the-art methods. For example, a layer of the model is deleted when an operator (not *None*) is replaced by the operator *None*. Similarly, a layer is added when an operator *None* is replaced by the other operators. On the other hand, in state-of-the-art methods, only one or few layers can be modified in one mutation. In our mutation, many layers can be modified

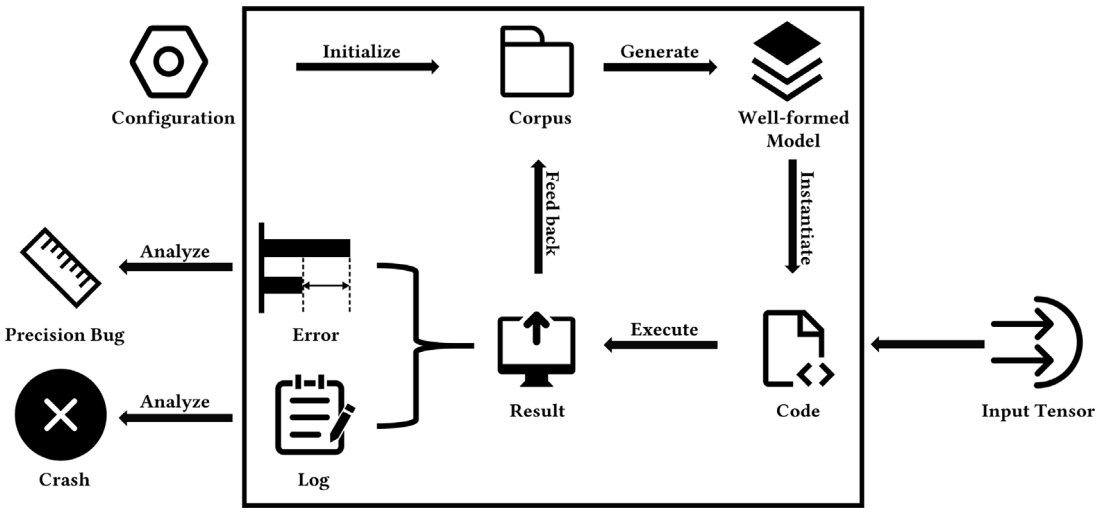

Fig. 4. Overview of Ramos.

simultaneously by replacing an operation combination. This advantage leads to a larger mutation amplitude, thus improving the diversity of models and the ability of models to trigger precision bugs.

4. Methodology

4.1. Overview

In this work, we propose Ramos, a hierarchical and heuristic method to test the DL framework. Fig. 4 shows the workflow of Ramos. Firstly, we propose a mutation-based corpus initialization method to generate various corpus according to the configuration (Section 4.2). Then, we propose a hierarchical and heuristic model generation method to generate well-formed models based on the generated corpus (Section 4.3). To perform differential testing, we propose an API mapping rule and apply it to transform models into code under different frameworks (Section 4.4). After that, we run the code under different frameworks and analyze the result to detect framework bugs (Section 4.5). Finally, we feed back the results to the next round (Section 4.6).

4.2. Initialization

Because a diverse corpus contributes to generating diverse models, we aim to initialize a corpus with high diversity. Unlike the other methods, which give corpus directly, we design an initialization method based on the random mutation to increase the diversity of the corpus. In our method, the corpus comprises several seeds, each a hierarchical structure. The method is as follows.

(1) Enter the configuration. The configuration includes the shape of the hierarchical structure and the input tensor.

(2) Generate a trivial hierarchical structure according to the configuration. In the trivial hierarchical structure, every motif is an operator chain from source to sink composed of the operator *identity*. The *basic_weight* of each operator and the *composite_weight* of each motif is set to the same default values respectively. The *fitness* of the trivial hierarchical structure is 0.

(3) According to the random mutation in Section 3, mutate the trivial hierarchical structure and add the newly generated structure into the corpus.

(4) Repeat Step 3 until the corpus reaches the pre-specified scale.

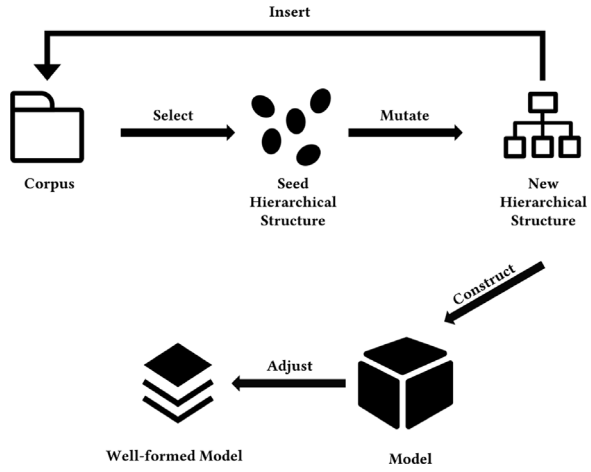

Fig. 5. Workflow of model generation in Ramos.

4.3. Model generation

Inspired by HNAS (Liu et al., 2017), Ramos uses a selection–mutation method to generate new models. Ramos chooses the most promising hierarchical structure from the corpus as seed, carries out heuristic mutation on the selected seed to generate a new hierarchical structure, and then builds a model according to the newly generated hierarchical structure. Finally, Ramos adjusts the new model to make it well-formed. Fig. 5 shows the workflow of the model generation.

(1) Select a seed. After initialization, the corpus is composed of several seeds. Each of them is a hierarchical structure. Ramos selects the most promising seed from the corpus, that is, picks the one that is more likely to construct a model with larger error after mutation.

Ramos uses *fitness* to measure the model error (Section 3). Notice that it is not a good idea to always choose the seed with the greatest *fitness*. Because this idea will cause seeds with less *fitness* never to be selected, which will reduce the diversity of seeds and thus reduce the diversity of models. To select the seed better, Ramos adopts a k-tournament algorithm (Fang and Li, 2010) based on *fitness* to select seeds from the corpus. Compared with selecting seeds with the largest *fitness* directly, the k-tournament algorithm adds randomness. This algorithm supports Ramos in

selecting the seeds with greater *fitness* and guarantees fairness at the same time. Therefore, more seeds have the chance to be selected so as to improve the variety of newly generated models. The process of the k-tournament algorithm is as follows.

Step 1. Select k groups from the corpus. As usual, the size of each group is set to 5% of the corpus size.

Step 2. Select hierarchical structures with top k *fitness* from k groups selected in Step 1, respectively.

Step 3. Select the ones with the top k *fitness* as seeds to be mutated from the $k \times k$ hierarchical structures selected in Step 2.

In addition, because Ramos only selects one seed from the corpus at a time, the value of k is 1.

(2) Mutate the seed. After selecting the most promising seed, Ramos mutates it to generate a new hierarchical structure. This hierarchical structure is used to generate a new model.

Ramos carries out heuristic mutation on the selected seed. The process of heuristic mutation is described in Section 3 in detail. After mutation, the hierarchical structure is added to the corpus as a seed for the next round.

(3) Construct a model. Based on the newly generated hierarchical structure, Ramos constructs a new model. The process is as follows.

Step 1. Ramos parses the hierarchical structure to the common part of the required cells. As described in Section 2.1, the required cells include normal cells and reduction cells. It is worth noting that there is a common part between normal cells and reduction cells. The common part of the cells can be generated by analyzing the newly generated hierarchical structure according to the following method.

Given the number of layers of the hierarchical structure is L. The motif in the highest layer is $o_L^{(1)}$. The vertex set of this motif is $V_M = \{v_{M1}, v_{M2}, v_{M3}, \ldots\}$ and the edge set is $E_M = \{e_{M1}, e_{M2}, e_{M3}, \ldots\}$ (see Section 3). For each edge $e_{Mk} \in E_M$, given the predecessor vertex is v_{Mi} and the successor vertex is v_{Mj}.

I. If e_{Mk} represents an operator, it remains unchanged.

II. If e_{Mk} represents an operator combination corresponding to a motif in the $L-1$ layer. Given this motif is N. The vertex set of this motif is V_N, and the edge set is E_N. Wherein, the source of this motif is v_{Nq}, and the sink is v_{Nw}. Firstly, add nodes in N to M. ($V_M \Leftarrow V_N \cup V_M$). Secondly, add edges in N to M. ($E_M \Leftarrow E_N \cup E_M$). Thirdly, add two directed edges that represent operator *identity*. One points from v_{Mi} to v_{Nq} and the other from v_{Nw} to v_{Mj}.

Remove layer $L-1$ after the above process is completed for each edge in E_M. This way, the hierarchical structure of L layers is converted to that of $L-1$ layers. Repeat the above process until there is one layer left in the hierarchical structure. The only motif left in the hierarchical structure is the common part of the required cells. It is worth noting that the common part is represented as a DAG.

Step 2. Ramos generates the required cells based on the common part generated in Step 1 by adding the different part between cells. In Ramos, the only difference between normal and reduction cells is the parameter setting of the last operator. Specifically, the *stride* of the last operator *Depthwise Convolution* is 1 in the normal cell and 2 in the reduction cell. After adding the operator with the correct parameter setting to the common part, Ramos generates a normal cell and a reduction cell. Then, Ramos clones the reduction cell to generate all required cells (including $\lambda - 1$ reduction cells and a normal cell)

Step 3. Ramos stacks the generated cells and thus constructs a new model. The method of stacking these cells is introduced in Section 2.1.

(4) Adjust the model. Some operators should be added to make the model well-formed. Firstly, the input tensor and output tensor of adjacent operators may be inconsistent in the number of channels. Secondly, the number of tensors expected to receive by each operator may need to be inconsistent with that actually receive. At last, *convolution* is not activated and normalized. Among them, the first two problems may cause model runtime failures, and the third problem may cause NaNs in the output. These NaNs are caused by bugs in DL models instead of DL frameworks. Therefore, the third problem may cause false positives.

To solve the first problem, Ramos records the channel number of the input tensor and output tensor of each operator and takes them as the parameter setting when calling API, so as to prevent the conflict of channel number.

To solve the second problem, when an operator (e.g., *Pooling*) can only receive one input tensor, but actually receives multiple input tensors, Ramos uses *Concatenate* in channel dimension to fuse the input tensors into one.

To solve the third problem, Ramos adds *Batchnorm* after all *Convolutions*, so as to ensure the rationality of tensor data distribution. In addition, Ramos adds an activation function after all *Convolutions*. Since a certain model usually contains only one type of activation function, all the activation functions added after *Convolution* are the same type. The type is selected randomly. Besides, according to the convention of DL model construction, the activation function mentioned above is added after *Batchnorm*.

In this section, Ramos generates a well-formed model represented by DAG. In the next section, Ramos will instantiate and execute this model and capture the output.

4.4. Model instantiation and execution

A new model is obtained in the model generation. As described in Section 2.2, the model should be instantiated under different frameworks to perform differential testing. Before instantiating, we need to determine the executing sequence and the channels of the operators. The former can be obtained by arranging the operators in the model according to the topological order of the predecessor vertexes. The latter can be counted as described in the model adjustment. As a result, the model can be transformed into code.

However, there is still a major challenge while instantiating the model. It is worth noting that there are differences in parameter settings between different frameworks. These differences may cause crashes and incorrect outputs, but they are not bugs in DL frameworks. These bugs should be avoided in DL framework testing. In order to prevent these bugs, we study the specific implementation of DL frameworks from the perspective of parameter settings. Based on our study, we conclude an API mapping rule which contains 15 mappings between different DL frameworks. These rules cover all types of operations (e.g., Convolution, Pooling, and Activation). We have published all of the API mappings on https://github.com/zyltt/JSS2022. The mapping can be divided into six categories.

Different padding modes. Different DL frameworks usually have different padding modes. For example, the padding mode "SAME" and "VALID" is not supported in PyTorch. The "SAME" mode in MindSpore is to fill zero evenly around the feature map. The purpose of the "SAME" pattern in TensorFlow is to make sure that the feature data is located in the center of feature map. Thus, it may fill zero evenly around the feature map, or only fill zeros on the right and bottom sides of the feature map. To make them consistent, Ramos adjusts the parameter settings related to padding when calling APIs.

Various parameter orders. The order of parameters in different DL frameworks varies. For example, when representing a tensor, the parameter order of TensorFlow is NHWC (batch, height, width, channel), and the channel is on the third dimension. However, the parameter order of Pytorch and MindSpore is NCHW (batch, channel, height, width), and the channel is on the

first dimension. When calling operators such as *Concatenate*, we need to set corresponding parameters in the corresponding order.

Various default values. The default value of the same parameter varies in different frameworks. If the default parameter setting is used when calling operators, the execution results will show huge differences. In order to solve this problem, we need to reassign same value to corresponding parameters when calling operators like *Batchnorm*.

Mismatched APIs. Some frameworks have unique parameters. To deal with these parameters, we study the specific meaning of the parameter and analyze the implementation of corresponding function, then determine the value of these parameters. For instance, when calling the operator *Average Pooling* under PyTorch, the parameter *count_include_pad* should be set to *False*, which means 0.0 is not involved in the computation. This parameter does not exist in other frameworks.

Different convolution kernels. In order to ensure the consistency of inference results, Ramos sets the same convolution kernel for all *Convolutions* under different DL frameworks.

Unimplemented APIs. If an operator is not implemented by any single API directly in the framework, Ramos will try to call another API to achieve the equivalent function with this operator. For example, only TensorFlow directly provides the API for the operator *Depthwise Convolution*. When *Depthwise Convolution* is required under PyTorch and MindSpore, Ramos will call the API for *Convolution*. (In the parameter setting, the channel number equals the group number.) In particular, it is worth noting that sometimes we need to call multiple APIs together. For instance, Ramos calls the API for the operator *Depthwise Convolution* and the operator *Convolution* sequentially to achieve the same function with the operator *Separable Convolution* under PyTorch and MindSpore.

The API mapping effectively avoids the potential problems caused by different parameter settings between different frameworks, thus reducing false positives. After instantiation, we give the input tensor and execute the code under different frameworks to obtain the result for further analysis. The source of the input tensor will be introduced in Section 5.

4.5. Result analysis

Crashes. Ramos keeps the log of model execution to detect crashes. A crash is often triggered with exception messages, which are recorded in the log of the model. So detailed information on crashes can be obtained by analyzing the log.

Precision bugs. Existing methods (Zhang et al., 2021b; Gu et al., 2022) usually set a threshold for the model error. When the model error exceeds this threshold, they consider that precision bugs are triggered. However, they set the value of the threshold based on subjective experience, so it is not appropriate. To solve this problem, we investigate the built-in test code of three mainstream frameworks, including TensorFlow, PyTorch, and MindSpore. We set the threshold as the maximum of all the built-in thresholds in all frameworks, which is used to judge if two tensors are absolutely equal. The value of the threshold ϵ is 1e−3.

4.6. Feedback

To increase the model error gradually, we design an error-oriented heuristic method. Because hierarchical structures, operators and operator combinations influence the model error differently, we design heuristic indicators for them respectively. Among these indicators, *Fitness* guides the selection of hierarchical structure in the next round. *Basic_weight* and *composite_weight* influence the selection of operators and operator combinations in the next round of mutation, respectively. The heuristic indicators *fitness*, *basic_weight* and *composite_weight* are modified as follows.

Firstly, Ramos calculates the model error. However, absolute error is impractical to obtain in automatic testing (Gu et al., 2022). As an alternative method, Ramos calculates *max_diff*. *Max_diff* represents the maximum relative error of the output tensor of the same model under two different frameworks with the same input. The formula of *max_diff* is:

$$max\_diff_{ij} = max(\left|output_i - output_j\right|)\ i, j \in \tau$$

where i, j represent two different frameworks. τ represents the set of frameworks to be tested. When finishing calculating all *max_diff* after executing the model, Ramos will update *fitness* in the newly generated hierarchical structure. The formula of *fitness* is:

$$fitness = avg(max\_diff_{ij})\ i, j \in \tau$$

Secondly, Ramos calculates the variation of *fitness* in each round, which is called $\Delta fitness$. $\Delta fitness$ reflects the influence of the hierarchical structures, operators and operator combinations on model error. Based on $\Delta fitness$, Ramos updates the *basic_weight* in the corpus and the *composite_weight* of each motif as follows.

Basic_weight describes the influence of a single operator. It is updated when a single operator is selected. For an operator o, when it has been selected v times, its weight is $basic\_weight_v$. The update formula of *basic_weight* corresponding to o when the operator is selected again is:

$$basic\_weight_{v+1} = basic\_weight_v + \Delta fitness$$

Composite_weight describes the influence of an operator combination. It is updated when an operator combination is selected. For an operator combination c, when it has been selected r times, its weight is $composite\_weight_r$. In initialization, we have set the initial value for the *composite_weight* (Section 4.2). The update formula of *composite_weight* corresponding to r when the operator is selected again is:

$$composite\_weight_{r+1} = composite\_weight_r + \Delta fitness$$

5. Evaluation

In order to evaluate the effectiveness and efficiency of the proposed method, we implement Ramos as a tool. We choose three state-of-the-art methods as baselines for evaluation, respectively LEMON (Wang et al., 2020b), Cradle (Pham et al., 2019), and Muffin (Gu et al., 2022). To get the replication code and results of each method, please visit https://github.com/zyltt/JSS2022.

5.1. Experimental setup

Experimental Environment. We run the experiment on a workstation. It is a GNU/Linux System with Ubuntu with kernel version 4.15.0-162-generic. It is equipped with Intel Core CPU Gold 5117 (32 cores, 2.0 GHz). The Python version is 3.9. We test three widely used DL frameworks, respectively TensorFlow, PyTorch, and MindSpore. The TensorFlow version is 2.6.0. The PyTorch version is 1.12.0, and the MindSpore version is 1.7.0.

Dataset. Ramos uses six widely used datasets as input tensors, including MNIST, F-MNIST, CIFAR-10, ImageNet, Sine-Wave, and Stock-Price. Sine-Wave contains the sine function value sequence, and Stock-Price contains the stock price sequence of Disneyland from 1997 to 2016. For the sake of fairness in comparison, Ramos covers all the datasets used by Muffin (Gu et al., 2022), LEMON (Wang et al., 2020b), and Cradle (Pham et al., 2019). In addition, similar to the other methods, Ramos supports randomly generated data as input tensors.

Table 1
Comparison of crash detection.

	Total	False positive	False positive rate
Ramos	15	0	0.0
Muffin	9	7	0.77
LEMON	5	1	0.2
Cradle	0	0	NaN

5.2. Research questions

In this paper, we aim to investigate the following research questions.

- **RQ1:** How does Ramos perform in detecting DL framework bugs?
- **RQ2:** How does Ramos perform in generating deep learning models?
- **RQ3:** How efficient Ramos is compared to state-of-the-art methods?
- **RQ4:** How do the different parameter settings affect the performance of Ramos?

In RQ1, we evaluate the effectiveness of Ramos in detecting different kinds of bugs, including crashes and precision bugs. In RQ2, we compare the models generated by different methods in two aspects, including the error and the variety. In RQ3, we compare the time overhead of different methods in model generation and test execution. In RQ4, we discuss the parameter settings in Ramos, and evaluate the influence of different parameter settings.

5.3. RQ1: Effectiveness of bug detection

5.3.1. Crashes

To evaluate the ability of Ramos to detect crashes, we download the source code of three state-of-the-art methods, respectively Cradle, LEMON, and Muffin. We run Ramos and these three methods under three wide-used frameworks, TensorFlow, PyTorch, and MindSpore. During model execution, we save the exception information in the log. These logs can provide helpful information for identifying crashes. When a crash is reported, two authors will scan the running log, analyze the cause of the crash, and determine whether it is a false positive. If the analysis results of the two authors are different, the authors will discuss together to reach a consensus. If there is still no consensus, the third author will refer to the analysis results and give the final conclusion.

We detect 15 different crashes in total. All of the crashes are reported to the community. 14 of them are confirmed, and another one is still in progress. 13 of the confirmed crashes are caused by incorrect memory allocation. For example, when *Sigmoid* is called, MindSpore fails to allocate memory for the output tensor properly and causes a crash. Another confirmed crash is because the operator *PReLU* does not support CPU. The bug being processed is caused by the failure of creating *Convolution* descriptor. Due to the complexity of this bug, developers need more time to confirm it. It is worth noting that this bug cannot be found by state-of-the-art methods. The diversity of generated models contributes to Ramos' success in detecting this bug.

We compare the number of detected crashes and the false positive rate between Ramos and other methods. The results are shown in Table 1. The first column of the table is the method name, and the last three columns are the total number of detected bugs, the number of false positives and the false positive rate, respectively.

On the one hand, the number of crashes successfully detected by Ramos is 15, which is the largest among all methods. On the other hand, the false positive rate of Ramos is significantly lower than that of other methods. Specifically, none of the bugs detected by Ramos are false positives. In contrast, 7 of the 9 bugs detected by Muffin are false positives and the false positive rate is 0.77. 1 of the 5 bugs detected by LEMON is false positive and the false positive rate is 0.2. We analyze the false positives in Muffin and LEMON and find that there are two main reasons. ❶ **Invalid parameter.** The parameters are set mistakenly. Specifically, the order of parameters is wrong. For example, many operators in TensorFlow only support tensors in NHWC (batch, height, width, channel) order. False positives are generated when Muffin provides tensors in NCHW (batch, channel, height, width) order for these operators. ❷ **Tensor shape conflicts.** The abuse of *reshape* by Muffin leads to the crash caused by tensor shape conflict. For example, when Muffin invokes the operator *self .fn*, Muffin provides a tensor with 33 channels. However, this operator expects to receive an operator with 32 channels. In Ramos, the API mapping rule prevents wrong parameter orders, and the model adjustment prevents the conflict of the tensor shape. We can figure from the results that the API mapping rule and model adjustment can effectively reduce the false positive rate.

5.3.2. Precision bugs

In this section, we aim to evaluate the ability of Ramos to detect precision bugs. We download all the models mentioned in Cradle that are still open source. We use Ramos, Muffin and LEMON to generate 100 models on each dataset, respectively. After that, we run these models and record the relative error of each model between different frameworks. When the error is larger than the threshold ϵ (1e−3, see Section 4.5), a precision bug is triggered.

When Muffin and LEMON instantiate the model, they do not consider the difference of parameter settings between different frameworks. Thus, they instantiate the model incorrectly. This mistake leads to different execution results of the same model in different frameworks. Muffin and LEMON record this difference as a part of the model error. Therefore, the error recorded by these two methods is wrong. To be fair, we use the model instantiation method in Ramos uniformly to instantiate the models generated by Cradle, Muffin, LEMON, and Ramos.

Table 2 shows the experimental results. The columns in the table represent the method name. The rows represent the dataset used. The numbers in the table represent the number of precision bugs detected. It is illustrated that Ramos detects a large number of precision bugs, while the other three methods do not detect any. Specifically, Ramos performs better in large datasets. For example, Ramos successfully detects 50 and 72 precision bugs under F-MNIST and Imagenet, respectively. While datasets such as Cifar-10, Sine-Wave and Stock-Price are relatively small, Ramos detects few precision bugs. However, it is still more than other methods. Experiments show that large datasets are more conducive to the accumulation of error, which leads to more serious precision errors and thus triggering more precision bugs. In conclusion, compared with the other methods, Ramos shows its outstanding ability in detecting precision bugs.

Answer to RQ1: Ramos can effectively detect crashes and precision bugs in DL frameworks, and the performance in precision bug detection is particularly excellent. Ramos detects 15 crashes in three DL frameworks, which is the most among all methods. In addition, Ramos successfully triggers the precision bugs in DL frameworks, which the other methods cannot detect.

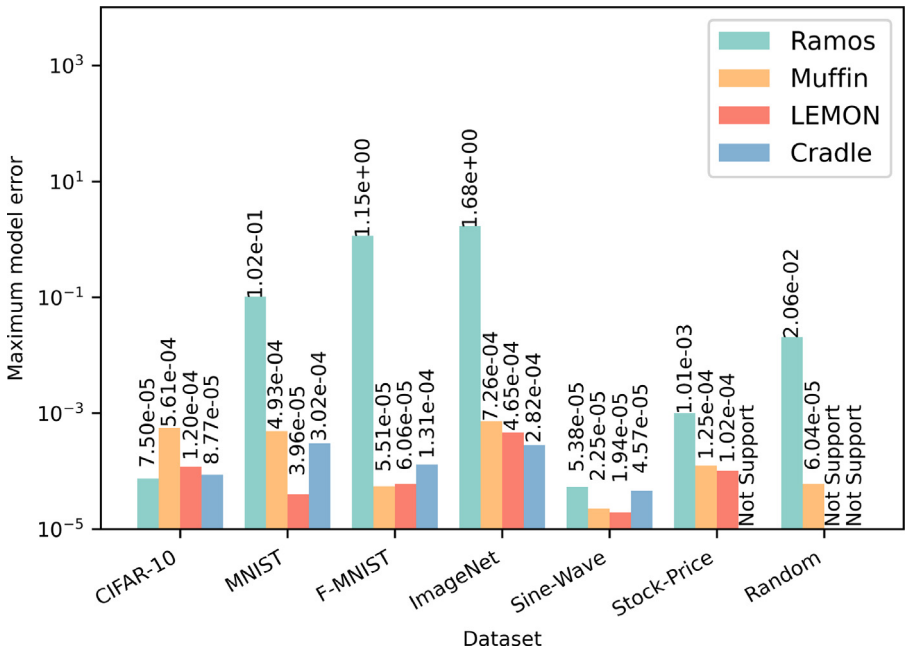

Fig. 6. The maximum model error.

Table 2
Comparison of precision bug detection.

	Ramos	Muffin	LEMON	Cradle
Cifar-10	0	0	0	0
MNIST	5	0	0	0
F-MNIST	50	0	0	0
ImageNet	72	0	0	0
Sine-Wave	0	0	0	0
Stock-Price	1	0	0	0
Random	26	0	0	0

5.4. RQ2: Effectiveness in model generation

Model generation is the most important component in DL framework testing. As described in Section 1, model error and model variety are the most significant requirements for generated models. The error and variety of generated models directly determine the bug detection effectiveness of the method. In this section, we evaluate the models generated by different methods in model error and model variety.

5.4.1. Model error

When detecting precision bugs, Ramos also records the maximum model error in each dataset. The experimental setup is exactly the same as the precision bug detection. The results are shown in Fig. 6. The horizontal axis of the bar graph represents the dataset used, and the vertical axis represents the maximum model error triggered. Four different colors are used in the figure to represent four different methods. The numbers in the graph represent the values of model error.

Except on Sine-Wave and Cifar-10, Ramos can trigger precision bugs successfully, and the maximum error is three magnitudes larger than ϵ (1e−3). It should be noted that the shape of the tensor in Sine-Wave and Cifar-10 is very small. The figure data involved in the computation is little, so it is reasonable that the error is slight. However, the error of models generated by the other methods is always less than ϵ, and they cannot trigger any precision bug. We can conclude the error triggered by models generated by Ramos is larger compared with the other methods. This advantage is related to the powerful heuristic strategy, which guides mutation in the direction of generating more models with larger error.

5.4.2. Model variety

In this section, we will compare the variety of models generated by Ramos, Muffin, and LEMON. The number of publicly available models that Cradle can use is less than 50, so it is not worth analyzing. We use Ramos, Muffin, and LEMON to generate 10,000 models separately and evaluate the model variety from two aspects, respectively external variety and internal variety.

External Variety. We calculate the edit distance between models to describe the external diversity of models. The edit distance describes the minimum number of modifications required for the DAG representation of one model to be modified to another model. When the model size is close, the larger the editing distance, the higher the external diversity of the model.

The experimental results show that the maximum edit distance of models generated by Ramos, Muffin and LEMON is 78.0, 29.0, 11.0, respectively. The average edit distance of models generated by Ramos, Muffin and LEMON is 16.41, 5.85, 0.06, respectively. It is obvious that both the average edit distance and the maximum edit distance in Ramos are the largest among these three methods. We can conclude that the models generated by Ramos have a higher external variety.

Internal Variety. We consider the model as a DAG. In this DAG, paths are related to the operators executed sequentially, and branches are related to multiple operators which are executed separately and then the execution results are fused. To evaluate the internal variety of models, we divide these models into two types, respectively chain-like model and non-chain-like model. The classification criteria are as follows.

We search for the longest path in the DAG and count the edges in this path. Furthermore, we calculate the proportion of the edges in the longest path to those in the DAG. We call this proportion chain-like rate. When the chain-like rate exceeds two-thirds, we regard this kind of model as chain-like model. The models with chain-like rate less than two-thirds are regarded as non-chain-like models.

The results show that among the models generated by Ramos, chain models are only 2219, 22%. 9087, 91% of the models generated by Muffin are chain-like models. All the models generated

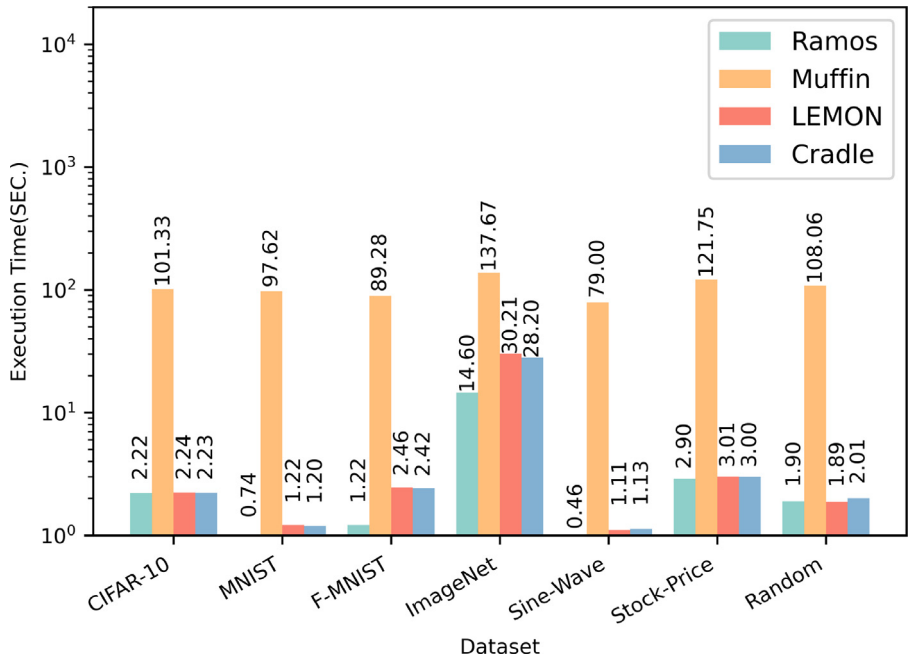

Fig. 7. Comparison of execution time (SEC.).

by LEMON are chain-like models. In conclusion, Ramos generates more non-chain-like models. So the internal variety of models generated by Ramos is higher.

In addition, we design an experiment to explore the relationship between chain-like models and precision bugs, we use the 10000 models generated by Ramos above as testing inputs to detect precision bugs, and then analyze the relevance between model structure and triggered precision bug. The experimental results show that among the 10000 models generated by Ramos, 2485 models trigger precision bugs successfully, including 230 chain-like models and 2255 models with other structures. It is obvious that most precision bugs are detected by models which are not chain-like. In conclusion, the improvement of the model variety is conducive to detecting precision bugs.

Answer to RQ2: Compared with the other methods, Ramos successfully generates models which can trigger a larger error. What is more, these models are with higher external and internal variety. The improvement in model error and model diversity is both helpful in detecting more DL framework bugs.

5.5. RQ3: Execution time

To evaluate the efficiency of Ramos, we generate 10000 models by Ramos, Muffin and LEMON respectively and record the total time for generating these models. The experimental results show that the total time of Ramos, Muffin and LEMON is 54.65 s, 224.57 s and 31.2 h. Because Cradle uses ready-made models as test inputs, it does not need to generate new models. Compared with Muffin and LEMON, the total execution time of Ramos for model generation is far shorter.

In addition, we compare the average execution time of each round of Ramos and three state-of-the-art methods on each dataset, as shown in Fig. 7. The horizontal axis represents the dataset used, and the vertical axis represents the execution time. Four different colors are used in the figure to represent four different methods. The numbers in the graph represent the time consumed by each method.

The results show that the execution time of Ramos is significantly shorter than that of Muffin in all datasets. The model execution time of Ramos, LEMON and Cradle are very close.

Answer to RQ3: Ramos takes the least time for both model generation and model execution. It indicates that Ramos remarkably reduces the overhead of computing resources.

5.6. RQ4: Parameter setting

In Ramos, there are three feedback modes and three operator type selection tendencies in mutation, respectively. The feedback modes are **BW**, **CW**, and **BCW**. In **BW**, only *basic_weight* is fed back. In **CW**, only *composite_weight* is fed back. In **BCW**, both *basic_weight* and *composite_weight* are fed back. The strategies of operator type selection tendency in mutation are **Single**, **Combination**, and **Same**. In **Single**, single operators tend to be selected. Specifically, the probability t of selecting single operators is set to 0.8. In **Combination**, operator combinations tend to be selected. Specifically, the probability t of selecting single operators is set to 0.2. In **Same**, the probability t of selecting single operators is set to 0.5. It should be noted that the two aspects mentioned above cannot independently affect the model error. For example, when the probability of selecting the primitive operator is set to 0.2 or 0.8, the impact on model error varies when setting the strategy of feedback mode as **BW**. These two aspects should be considered comprehensively. Therefore, we design experiments to evaluate different strategy pairs. Similar to other methods (Gu et al., 2022), we conduct this experiment using randomly generated data.

Fig. 8 shows the experimental results. The horizontal axis of the bar graph represents the operator type selection tendency and the vertical axis represents the maximum error. Three different colors are used in the figure to represent three feedback modes. The numbers in the graph represent the maximum error caused in the experiment of the corresponding strategy pair. It can be confirmed from the experimental results that the strategy pair composed of **BCW** and **Combination** is obviously better than

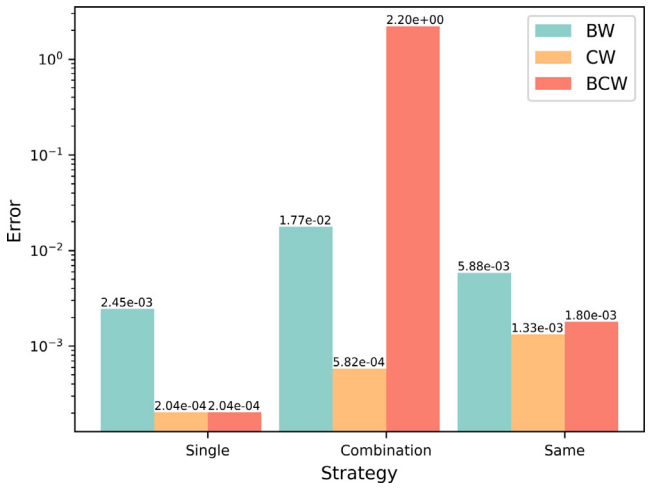

Fig. 8. Testing results with different strategy pairs.

other pairs. From the perspective of feedback strategy, **BCW** has the best overall effect. Among them, the effect of **BCW** is significantly better than that of **BW**, which shows that the feedback for *composite_weight* works. The effect of **BCW** is significantly better than that of **CW**, which shows that the feedback for *basic_weight* works. As to the strategy of operator type selection tendency in mutation, the results show a downward trend with the increase in the probability of selecting an operator. It can be seen that the mutation selecting operator combinations can also effectively magnify the error.

Answer to RQ4: Compared with other strategy pairs, the strategy pair composed of **BCW** and **Combination** is the most effective in detecting precision bugs. Besides, both the feedback for *basic_weight* and *composite_weight* are very effective, and the effect is much better when the two kinds of feedback are combined together. What is more, the mutation selecting operator combinations also contributes to magnifying the model error.

6. Validity & threat

The internal threat lies in the establishment of the API mapping rule, which involves subjective factors. In order to reduce their negative impact, we refer to the official documents of the relevant DL framework. We confirm the vague description in the document by researching the source code of the DL framework.

The external threat lies in the dataset used. In order to diversify the experimental results, we use six popular datasets. Moreover, similar to Muffin (Gu et al., 2022), Ramos is also not sensitive to the dataset. So Ramos is also valid when using the other datasets. In addition, in order to make our method general, the frameworks we choose are the most widely used. The core idea of Ramos can also be applied to test other DL frameworks.

7. Related work

7.1. NAS techniques

NAS (Neural Architecture Search) is a model construction technology in the artificial intelligence field. Hieu Pham et al. (2018) propose an algorithm called ENAS to reduce computation by sharing weights among DL models. Xie and Yuille (2017) utilize a genetic algorithm to calculate the fitness function so as to search for the best model. Liu et al. (2018) transform the model structure search into the optimization problem of continuous space. In addition, Negrinho and Gordon (2017) introduce the Monte Carlo tree search to enhance NAS.

However, the amount of computation of NAS in the search process is still huge. In order to solve this problem, researchers introduce a variety of HNAS (Hierarchical NAS) methods. Liu et al. (2017) design a hierarchical structure to generate models. Xia and Ding (2020) adopt a progressive search strategy of operator pruning, which enables the model to design the operator of each layer automatically.

Unfortunately, the NAS methods mentioned above are designed to search for models with better performance, rather than effectively trigger DL framework bugs. So it cannot be directly applied to DL framework testing. We combine the heuristic algorithm and HNAS technology and take the model error as the heuristic indicator.

7.2. Deep learning framework testing

Research shows there are bugs in DL frameworks (Zhang et al., 2018; Islam et al., 2019, 2020; Humbatova et al., 2020; Jia et al., 2020; Anon., 2021; Du et al., 2022; Yang et al., 2022; Wang et al., 2020a). The quality of the DL model is highly dependent on that of the DL framework, so DL framework testing is very important. The existing DL framework testing methods mainly include interface testing of single operators and testing with DL models as input.

Interface testing of single operators takes tensor as testing input to test the single operator interface provided by DL framework one by one. Zhang et al. propose Predoo (Zhang et al., 2021b), a fuzz testing method for testing precision bugs. Zhang et al. (2021a) propose a differential fuzz testing method to evaluate implementation bugs, execution time, and outputs of single operators. Xie et al. (2022) use automated technology to analyze the API documents of the DL framework and extract the constraints declared in the documents to generate valid testing inputs. Deng et al. (2022) obtain all possible candidate APIs by automatically inferring the relationship between APIs, and then perform fuzz testing.

However, some DL framework bugs can only be triggered when the model is executing. That is to say, these bugs cannot be detected by interface testing of single operators. To solve this problem, DL framework testing with DL models as input is proposed. CRADLE (Pham et al., 2019) executes the ready-made DL models to detect low-level DL frameworks. LEMON (Wang et al., 2020b) proposes a variety of mutation rules to generate models as testing input by mutating on ready-made models. Audee (Guo et al., 2020) introduces genetic algorithm into model search. Luo et al. (2021) design a graph-based fuzz testing method and summarizes six kinds of mutation. Li et al. (2022) propose a new mutation technique which introduces the Markov chain Monte Carlo algorithm to explore new layer types, layer pairs, and layer parameters. Gu et al. (2022) randomly generate the structural information of the model based on the templates and select operators layer by layer according to the model structure.

Different from the methods mentioned above, we design a novel hierarchical structure to support the mutation at the operator combination level. What is more, we design a heuristic method based on the model error.

8. Conclusion

We propose Ramos, a novel hierarchical heuristic DL framework testing method to detect crashes and precision bugs. Experimental results show that Ramos can detect crashes and precision bugs effectively. Specifically, thanks to the proposed heuristic indicators, Ramos has an extraordinary performance far beyond other methods in the detection of precision bugs. Then, Ramos designs a hierarchical structure, which increases the amplitude of mutation, so Ramos manages to generate more various models. Finally, the model adjustment and API mapping rule proposed by Ramos effectively reduce false positives. Ramos provides a new idea for model generation in DL framework testing. In future work, we will continue to optimize the model generation method.

CRediT authorship contribution statement

Yinglong Zou: Conceptualization, Methodology, Software, Writing – original draft. **Haofeng Sun:** Software, Writing – original draft. **Chunrong Fang:** Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Jiawei Liu:** Conceptualization, Investigation. **Zhenping Zhang:** Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This research is partially supported by the National Natural Science Foundation of China (No. 61932012, 62141215, 62272220) and Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD2020061710300 1003), and National Undergraduate Training Program for Innovation and Entrepreneurship (202210284097Z). Yinglong Zou and Haofeng Sun contribute equally to this research.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al., 2016. TensorFlow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation. pp. 265–283.

Anon., 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. J. Syst. Softw. 177, 110935.

Chen, Z.-M., Wei, X.-S., Wang, P., Guo, Y., 2019. Multi-label image recognition with graph convolutional networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 5177–5186.

Chowdhary, K., 2020. Natural language processing. Fundam. Artif. Intell. 603–649.

Deng, Y., Yang, C., Wei, A., Zhang, L., 2022. Fuzzing deep-learning libraries via automated relational API inference. arXiv preprint arXiv:2207.05531.

Du, X., Sui, Y., Liu, Z., Ai, J., 2022. An empirical study of fault triggers in deep learning frameworks. IEEE Trans. Dependable Secure Comput..

Fang, Y., Li, J., 2010. A review of tournament selection in genetic programming. In: International Symposium on Intelligence Computation and Applications. Springer, pp. 181–192.

Google, 2015. Tensorflow. https://www.tensorflow.org/. (Accessed 2022).

Gu, J., Luo, X., Zhou, Y., Wang, X., 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. arXiv preprint arXiv:2204.08734.

Guo, Q., Xie, X., Li, Y., Zhang, X., Liu, Y., Li, X., Shen, C., 2020. Audee: Automated testing for deep learning frameworks. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 486–498.

He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep residual learning for image recognition. CoRR, arXiv:1512.03385.

Huawei, 2019. Mindspore. https://www.mindspore.cn/. (Accessed 2022).

Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P., 2020. Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1110–1121.

Islam, M.J., Nguyen, G., Pan, R., Rajan, H., 2019. A comprehensive study on deep learning bug characteristics. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 510–520.

Islam, M.J., Pan, R., Nguyen, G., Rajan, H., 2020. Repairing deep neural networks: Fix patterns and challenges. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. IEEE, pp. 1135–1146.

Ji, H., Liu, Z., Yan, W.Q., Klette, R., 2019. Early diagnosis of alzheimer's disease using deep learning. In: Proceedings of the 2nd International Conference on Control and Computer Vision. Association for Computing Machinery, New York, NY, USA, pp. 87–91. http://dx.doi.org/10.1145/3341016.3341024.

Jia, L., Zhong, H., Wang, X., Huang, L., Lu, X., 2020. An empirical study on bugs inside tensorflow. In: International Conference on Database Systems for Advanced Applications. Springer, pp. 604–620.

Ju, R., Hu, C., Zhou, P., Li, Q., 2019. Early diagnosis of alzheimer's disease based on resting-state brain networks and deep learning. IEEE/ACM Trans. Comput. Biol. Bioinform. 16 (1), 244–257. http://dx.doi.org/10.1109/TCBB.2017.2776910.

Li, M., Cao, J., Tian, Y., Li, T.O., Wen, M., Cheung, S.-C., 2022. Memo: Coverage-guided model generation for deep learning library testing. arXiv preprint arXiv:2208.01508.

Li, P., Zhao, H., Liu, P., Cao, F., 2020. RTM3D: Real-time monocular 3D detection from object keypoints for autonomous driving. In: European Conference on Computer Vision. Springer, pp. 644–660.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K., 2017. Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436.

Liu, H., Simonyan, K., Yang, Y., 2018. Darts: Differentiable architecture search. arXiv preprint arXiv:1806.09055.

Luo, W., Chai, D., Ruan, X., Wang, J., Fang, C., Chen, Z., 2021. Graph-based fuzz testing for deep learning inference engines. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. IEEE, pp. 288–299.

Meta, 2016. Pytorch. https://pytorch.org/. (Accessed 2022).

Negrinho, R., Gordon, G., 2017. Deeparchitect: Automatically designing and training deep architectures. arXiv preprint arXiv:1704.08792.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al., 2019. Pytorch: An imperative style high-performance deep learning library. Adv. Neural Inf. Process. Syst. 32.

Pham, H., Guan, M., Zoph, B., Le, Q., Dean, J., 2018. Efficient neural architecture search via parameters sharing. In: International Conference on Machine Learning. PMLR, pp. 4095–4104.

Pham, H.V., Lutellier, T., Qi, W., Tan, L., 2019. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. IEEE, pp. 1027–1038.

Rao, Q., Frtunikj, J., 2018. Deep learning for self-driving cars: Chances and challenges. In: Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems. Association for Computing Machinery, New York, NY, USA, pp. 35–38. http://dx.doi.org/10.1145/3194085.3194087.

Shatnawi, A., Al-Bdour, G., Al-Qurran, R., Al-Ayyoub, M., 2018. A comparative study of open source deep learning frameworks. In: 2018 9th International Conference on Information and Communication Systems. IEEE, pp. 72–77.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2014. Going deeper with convolutions. CoRR, arXiv:1409.4842.

Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E., 2018. Deep learning for computer vision: A brief review. Comput. Intell. Neurosci. 2018.

Wang, C., Shen, J., Fang, C., Guan, X., Wu, K., Wang, J., 2020a. Accuracy measurement of deep neural network accelerator via metamorphic testing. In: IEEE International Conference on Artificial Intelligence Testing, AITest. pp. 55–61.

Wang, Z., Yan, M., Chen, J., Liu, S., Zhang, D., 2020b. 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. pp. 788–799.

Wei, A., Deng, Y., Yang, C., Zhang, L., 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. arXiv preprint arXiv:2201.06589.

Xia, X., Ding, W., 2020. HNAS: Hierarchical neural architecture search on mobile devices. arXiv preprint arXiv:2005.07564.

Xie, D., Li, Y., Kim, M., Pham, H.V., Tan, L., Zhang, X., Godfrey, M.W., 2022. Docter: Documentation-guided fuzzing for testing deep learning API functions. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 176–188.

Xie, L., Yuille, A., 2017. Genetic cnn. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 1379–1388.

Yang, Y., He, T., Xia, Z., Feng, Y., 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. Inf. Softw. Technol. 151, 107004.
Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., Zhang, L., 2018. An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 129–140.
Zhang, X., Liu, J., Sun, N., Fang, C., Liu, J., Wang, J., Chai, D., Chen, Z., 2021a. DUO: Differential fuzzing for deep learning operators. IEEE Trans. Reliab. 70 (4), 1671–1685.
Zhang, X., Sun, N., Fang, C., Liu, J., Liu, J., Chai, D., Wang, J., Chen, Z., 2021b. Predoo: Precision testing of deep learning operators. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 400–412.
Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., Deng, M., Li, H., 2019. T-GCN: A temporal graph convolutional network for traffic prediction. IEEE Trans. Intell. Transp. Syst. 21 (9), 3848–3858.

Yinglong Zou is currently a B.E. candidate in software engineering with the State Key Laboratory for Novel Software Technology at Nanjing University, Jiangsu. His research interests include in testing deep learning libraries, automated testing, etc.

Haofeng Sun is currently a B.E. candidate in software engineering at the Software Institute of Nanjing University, Jangsu, China. His research interests lie in DL framework testing, automated testing, crowdsourced testing.

Chunrong Fang received the B.E. and Ph.D. degrees in software engineering from Software Institute, Nanjing University, Jiangsu, China. He is currently an assistant professor with the Software Institute of Nanjing University. His research interests lie in intelligent software engineering, e.g. BigCode and AITesting.

Jiawei Liu received the B.E. and M.E. degrees in software engineering in 2019 and 2021, respectively. She is currently working toward the Ph.D. degree in software engineering with the State Key Laboratory for Novel Software Technology, Nanjing University, Jiangsu, China. Her research interests include in testing deep learning libraries, evaluation of data quality, data augmentation, etc.

Zhenping Zhang received the B.E. degree in software engineering from Software Institute, Nanjing University, Jiangsu, China. He is currently a Master degree candidate in software engineering at the Software Institute of Nanjing University. His research interests lie in software testing and neural network testing.