2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00037

Graph-based Fuzz Testing for Deep Learning Inference Engines

Weisi Luo†, Dong Chai†, Xiaoyue Run†, Jiang Wang†, Chunrong Fang*, Zhenyu Chen*
†HiSilicon, Huawei, China
*State Key Laboratory for Novel Software Technology, Nanjing University, China
Corresponding author: zychen@nju.edu.cn

***Abstract*—With the wide use of Deep Learning (DL) systems, academy and industry begin to pay attention to their quality. Testing is one of the major methods of quality assurance. However, existing testing techniques focus on the quality of DL models but lacks attention to the core underlying inference engines (i.e., frameworks and libraries). Inspired by the success stories of fuzz testing, we design a graph-based fuzz testing method to improve the quality of DL inference engines. This method is naturally followed by the graph structure of DL models. A novel operator-level coverage criterion based on graph theory is introduced and six different mutations are implemented to generate diversified DL models by exploring combinations of model structures, parameters, and data inputs. The Monte Carlo Tree Search (MCTS) is used to drive DL model generation without a training process. The experimental results show that the MCTS outperforms the random method in boosting operator-level coverage and detecting exceptions. Our method has discovered more than 40 different exceptions in three types of undesired behaviors: model conversion failure, inference failure, output comparison failure. The mutation strategies are useful to generate new valid test inputs, by up to an 8.2% more operator-level coverage on average and 8.6 more exceptions captured.**

***Index Terms*—Deep Learning Inference Engine, Graph Theory, Deep Learning Models, Operator-Level Coverage, Monte Carlo Tree Search**

I. Introduction

Deep Learning (DL) is a popular method for hard computational problems in various domains, such as image classification [1] and speech recognition [2]. There are almost innumerable combinations of DL frameworks, data sets, models, platforms, and so on [3]. On the hardware side, the platforms are tremendously diversified, ranging from common processors (e.g., CPUs, GPUs and NPUs) to FPGAs, ASICs, and exotic accelerators such as analog and mixed-signal processors. These platforms come with hardware-specific features and constraints that enable or disrupt inference depending on DL models and scenarios. On the software side, a number of DL inference engines invoked by DL applications commonly serve for optimizing various DL models and performing run-time acceleration inference on the above devices, such as NVIDIA TensorRT [4], TensorFlow-Lite [5] and Alibaba MNN [6]. Hence, the quality of DL inference engines supporting DL models is important to the quality of applications. To the best of our knowledge, there is still a lack of systematic testing methods for DL inference engines.

Fuzz testing is a widely used automated testing technique that generates random data as inputs to detect crashes, memory leaks, and other failures in software [7]. Fuzz testing has been shown to be a promising direction for quality assurance of DL systems [8] [9]. However, the existing fuzz testing techniques depend heavily on manually designed DL models, and usually perform perturbations, e.g., layer addition, layer removal, data shuffle, noise perturbation on such existing models or data [10] [11] [12]. Different from fuzz testing on DL models, fuzz testing on DL inference engines is expected to generate diversified DL models by exploring combinations of model structures, parameters, weights and inputs. It is very challenging to generate such complex graph structured data automatically and effectively.

The first challenge in fuzz testing of DL inference engines is to generate diversified DL models to trigger different structured parts of a given DL inference engine. These structured parts include model structure conversion, model optimizations (e.g., operator fusion), data format conversion (e.g., NCHW, NHWC, NC4HW4 [6]), operator implementation, calculation graph scheduling, data-movement a hierarchy of tiles, and so on [13]. The second challenge is to capture the behaviors of each test, such that fuzz testing can be well directed in generating new tests. The existing neural coverage criteria of DL models cannot work in this testing scenario, because the inputs for testing DL inference engines are DL models. A novel criterion is required to capture behaviors of DL inference engines rather than DL models. It is natural to design testing methods inspired by the graph structure of DL models and analyze the behaviors of the DL inference engine under test against different DL models.

In this paper, a novel graph-based fuzz testing method is designed to test DL inference engines via generating DL models as digraphs. The idea of this method naturally conforms to the graphic structure of the DL model. It alleviates the problem of generating a large number of diverse DL models. Since DL models are not a simple digraph, they have rich characteristics of deep learning elements. Thus, beyond the basis of DL model generation, four model-level mutations, including graph edges addition, graph edges removal, block nodes addition, block nodes removal, and two source-level mutations, including tensor shape mutation and parameter mutation, are proposed to generate more diversified DL models effectively.

To guide more effective fuzz testing for a given DL inference engine under test, the dynamic behaviors of each test (a

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00037

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

DL model) should be captured. The operator-level coverage criterion is introduced from graph theory to measure the parts of a DL inference engine's logic exercised by a test set (a number of DL models) based on operator types, structures (e.g., input degree, output degree and single edges from graph theory), tensor shapes and parameters. The success ratio and operator-level coverage of an operator are used as feedback to the Monte Carlo Tree Search (MCTS). MCTS is used to solve the search problem to determine whether an operator is select or not in a new model, such that the most promising blocks can be chosen to generate stochastic DL models.

The experiments have been designed and conducted with MNN [6] on X86 CPU and ARM CPU. The experimental results shows that our method is effective in detecting exceptions of DL inference engine, with more than 40 different exceptions discovered, such as crashes, inconsistencies, nan and inf bugs. Besides, the MCTS-based search performs better than the random-based search in boosting operator-level coverage(6.7% more) and detecting exceptions (9.7 more). Furthermore, the mutation strategy helps produce 8.2% more operator coverage and 8.6 more exceptions detected on average.

Our main contributions in this paper are as follows.

- A novel graph-based fuzz testing technique is proposed for DL inference engines, where DL models are defined as digraphs from a natural and effective basis.
- A novel operator-level coverage criterion is introduced to enhance fuzz testing via a reward-guided metric, which can estimate the amount of DL inference engine's logic explored.
- Some graph-based model-level mutations (graph edges addition, graph edges removal, block nodes addition, block nodes removal) and source-level mutations (tensor shape mutation, parameter mutation) are proposed to generate diversified DL models.
- An experimental evaluation on an industrial inference engine is conducted. The results show that the operator-level coverage guided testing framework improves the effectiveness in detecting exceptions.

More details of graph-based fuzz testing and the experiments can be found at https://github.com/gbftdlie/Graph-based-fuzz-testing.

II. Background

A. Workflow of DL inference engines

DL inference engines are invoked by DL applications to load and run models on devices with inference hardware accelerators. The workflows of existing inference engines are similar. Take MNN [6](a lightweight DL inference engine developed by Alibaba Inc) as an example, as shown in Fig. 1, the inference workflow can be roughly divided into two phases: (1) Conversion: phase of converting those training framework models (e.g., TensorFlow (Lite), Caffe, and ONNX) into MNN models and optimizing DL models by operator fusion, operator substitution, and layout adjustment. Furthermore, MNN models can be quantized optionally. (2) Inference: phase of loading MNN model and inferring. The interpreter of MNN consists of engine and backends. The former is responsible for loading a MNN model and scheduling a computational graph; the latter includes the memory allocation and the operator implementation under each computing device.

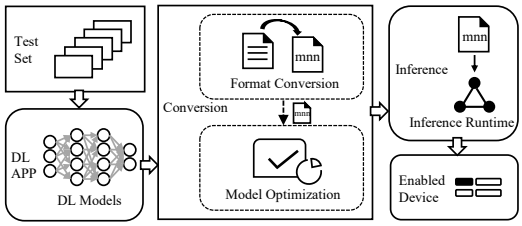


Fig. 1. Inference workflow with MNN

B. Limitations of existing testing techniques

Fuzz testing has been proved to be effective in exploring the input space of DL system testing. An important part of fuzz testing is how to feedback. DeepXplore [11] introduced the concept of neuron coverage for measuring the parts of a DL system's logic exercised by a set of test inputs based on the number of neurons activated (i.e., the output values are higher than a threshold) by the inputs. Enlightened by DeepXplore, a number of different coverage criteria have been applied to DL testing, such as TensorFuzz [10], Deeptest [14] and Deepgauge [15]. These testing techniques focus on testing the quality of specific DL models. However, in DL inference engine testing, when the input model is changed, the coverage will be invalid as feedback.

Another important part of fuzz testing is to generate test inputs via mutating a given set of inputs. DeepMutation [12] proposed a set of mutation operators, including changing weights, biases and inputs via disturbing, exchanging neurons within a layer, adding or removing a layer of which input and output dimensions are equal, like batch normalization, etc. This method mutates a single model to simulate error scenarios. TensorFuzz is another fuzz test generation tool based on input data mutation for a DL model. These methods can also effectively guarantee the quality of specific DL models. But they cannot generate a large number of diversified models for testing inference engines, that is, despite a large amount of input generated, the given model has not changed or changed very little. Therefore they are unlikely to detect erroneous behaviors and trigger different parts of a DL inference engine's logic.

In general, existing testing techniques focus on the quality of DL models, but lack attention to testing DL inference engines. Diversified combinations of operators (models) are more capable of triggering DL inference engine issues than an specific combination of operators (a specific/single model). The issues triggered by a single model are limited. We need to generate a large number of models by combining operators as the test inputs of DL inference engines .

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

III. METHODOLOGY

In this section, we provide detailed technical descriptions of graph-based fuzz testing for DL inference engines.

A. Definitions

The stochastic networks generation involves the following definitions: **digraph**, **subgraph**, **block**, **block corpus**, **mutation action**.

Digraph. Graph theory provides an excellent framework for studying and interpreting neural network topologies [16]. A neural network can be denoted by a digraph $G = (V, E)$. V is a set of operators (e.g., Conv2d, Relu and Softmax). $E \subseteq \{(x, y)|(x, y) \in V^2 \bigwedge x \neq y\}$ is a set of directed edges which are ordered pairs of distinct operators (e.g., x and y). In neural networks, edges are data flows.

Subgraph. From the introduction above, some specified structures of DL models will be specially processed (e.g., operator fusion and replacement [17] [18] to run a faster inference). There is a very low probability that these specified structures could be generated randomly. Thus subgraphs are applied to blocks to define those specified structures directly in testing. Formally, digraph $G' = (V', E')$ is a subgraph of G iff $V' \subseteq V, E' \subseteq E \bigwedge((x, y) \in E' \rightarrow x, y \in V')$. x and y are two distinct operators.

Block. Subgraphs or operators of a neural network are defined as blocks in this paper. A network is constructed by operators and subgraphs as shown in Fig. 2.

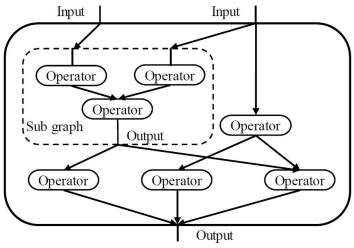


Fig. 2. A network is constructed by operators and subgraphs.

Block corpus. A block corpus contains blocks to be chosen and their attributes, including block name, allowed range of in-degree and out-degree, inner edges of the block. Inner edges are required when the block is a subgraph and can be empty otherwise. To construct the block corpus, the tester need to first confirm the types of operators and subgraphs to be tested, and then fill in their above attributes into the block corpus.

Mutation action. Let $I_1, I_2, \ldots, I_n$ be a sequence of mutated test sets, where I_k is generated by k-th mutation action $MA(bs_k, ms_k)$. bs_k and ms_k are the blocks selection and mutation operators selection in k-th mutation action respectively. A tuple of the two actions forms a complete action $MA(bs, ms)$.

B. Operator-level Coverage Criterion

Traditional code coverage criteria are ineffective in DL testing. As discussed in Section II.B, recently proposed neuron coverage criteria are still invalid, as DL inference engine testing involves different models. A novel operator-level criterion is proposed to capture differences in models combined by operators, which provide feedback to guide the proposed graph-based fuzz testing to generate diversified models. As defined in III-A, we use the model structures, input tensor shapes, parameters to characterize behaviors of operators in DL models. As defined in III-A, we use the input degree (the number of input data flows), the output data flows (the number of output data flows), input tensor shapes (NHWC, etc.) and parameters (e.g., dilation of Conv2d) of operators to characterize behaviors of operators in DL models.

Given a block corpus BC and a test set I, operator-level coverage criterion is defined as follows:

Operator Type Coverage(OTC). Let n_t be the number of total types of operators defined in BC. Let $OTC\_op(c, I)$ be 1 when the operator c is in the I, and be 0 otherwise. The OTC of I is defined as :

$$OTC(I) = \frac{\sum OTC\_op(c, I)}{n_t} \tag{1}$$

Input Degree Coverage(IDC). Let n_{id_c} be the total number of different input degrees of operator c in BC. Let $f_{id\_op}(I)$ be the number of different input degrees for operator c in I. The input degree coverage of operator c is defined as the ratio of $f_{id\_op}(I)$ to n_{id_op}: $IDC\_op(c, I) = \frac{f_{id\_op}(I)}{n_{id\_c}}$.

$$IDC(I) = \frac{\sum IDC\_op(c, I)}{n_t} \tag{2}$$

Output Degree Coverage(ODC). Let n_{od_c} be the total number of different output degrees of operator c in BC. Let f_{od_op} (I) be the number of different output degrees of operator c in I. The output degree coverage of operator c is defined as the ratio of $f_{od\_op}(I)$ to n_{od_c}: $ODC\_op(c, I) = \frac{f_{od\_op}(I)}{n_{od\_c}}$. The ODC of I is defined as:

$$ODC(I) = \frac{\sum ODC\_op(c, I)}{n_t} \tag{3}$$

Single Edge Coverage (SEC). Let $f_{se}(c, I)$ be the number of different edges that directed from the operator c to others in I. The number of total edges that directed from the operator c to others in BC is n_t. The single edge coverage of operator c is defined as the ratio of $f_{se}(c, I)$ to n_t: $SEC\_op(c, I) = \frac{f_{se}(c,I)}{n_t}$. The SEC of I is defined as:

$$SEC(I) = \frac{\sum SEC\_op(c, I)}{n_t} \tag{4}$$

Shapes&Parameters Coverage(SPC). Let n_{maxspc} be the expected maximum of shape&parameter. Let $f_{spc}(c, I)$ be the number of distinct vectors including tensor shapes and parameters for operator c in I. The SPC of operator c in I is defined as: $SPC\_op(c, I) = \frac{f_{spc}(c,I)}{n_{maxspc}}$. The SPC of I is defined as:

$$SPC(I) = \frac{\sum SPC\_op(c, I)}{n_t} \tag{5}$$

Operator-level Coverage (OLC). Let OLC of the

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

operator c be the weighted mean of a set of metrics $Z\_op = \{OTC\_op(c,I), IDC\_op(c,I), ODC\_op(c,I), SEC\_op(c,I), SPC\_op(I)\}$ with corresponding non-negative weights $\{w\_op_1, w\_op_2, ..., w\_op_5\}$. OLC of operator c is defined as:

$$OLC\_op(c,I) = \frac{\sum w\_op_i m\_op_i}{\sum w\_op_i}, m\_op_i \in Z\_op \quad (6)$$

Let OLC of the test set I be the weighted mean of a set of metrics $Z = \{OTC(I), IDC(I), ODC(I), SEC(I), SPC(I)\}$ with corresponding non-negative weights $\{w_1, w_2, ..., w_5\}$. Formally, OLC of test set I is defined as:

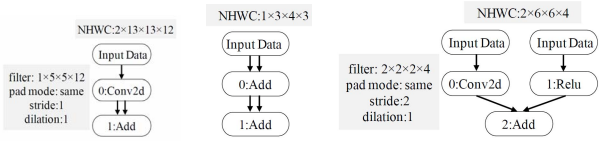
$$OLC(I) = \frac{\sum w_i m_i}{\sum w_i}, m_i \in Z \quad (7)$$

Some weights may be zero. For example, the weight of $ODC(I)$ should be 0 when expected output degree of operators are all 1 in test samples of test set I.

For example, Fig. 3 shows three NNs in test set I generated by block corpus BC in TABLE I. Tensor format is NHWC. Three blocks Conv2d, Relu and Add, and their input and output degree, are defined. In operator-level coverage, the n_{maxspc} of Formula(5) is set to 10. Operator-level coverage result for each operate and test set I are calculated and listed in TABLE II respectively.

TABLE I
BLOCK CORPUS OF TEST SET I

Block Name	Input Degree	Output Degree	Inner Edges
Conv2d	{1}	{ 0,1,2}	N/A
Relu	{1}	{ 0,1,2}	N/A
Add	{2}	{ 0,1,2}	N/A



(a) NN1 (b) NN2 (c) NN3

Fig. 3. Three NNs in Test Set I

TABLE II
OPERATOR-LEVEL COVERAGE OF EACH OPERATOR AND TEST SET I

Object	OTC	IDC	ODC	SEC	SPC	OLC
Conv2d	100%	100%	66.7%	33.3%	20%	64%
Relu	100%	100%	33.3%	33.3%	10%	55.3%
Add	100%	100%	66.7%	33.3%	30%	66%
I	100%	100%	55.6%	33.3%	20%	61.8%

C. Framework

The core idea of graph-based fuzz testing is to maximize operator-level coverage on a DL inference engine such that as many erroneous behaviors as possible can be exposed. A large number of test samples (i.e., models) can be constructed by mutating existing DL models (modeled as graphs) and operator-level coverage is used as feedback to guide test input generation.

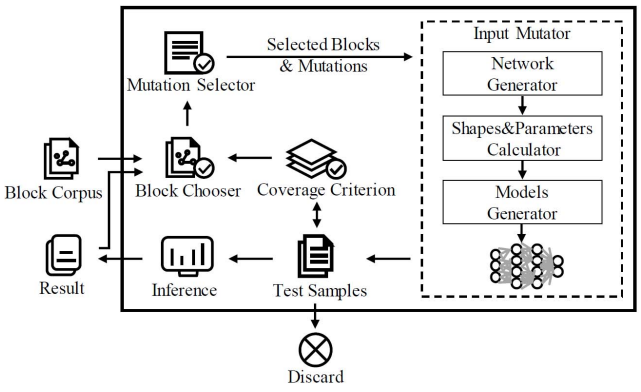


Fig. 4. Framework of graph-based testing

Framework. As depicted in Fig. 4, the graph-based fuzz testing framework is composed of block chooser, coverage criterion, input mutator, and mutation selector. For each iteration, the MCTS-based block chooser chooses a set of blocks b from block corpus. The mutation selector chooses one or more mutations scholastically to determine mutating rules m. Parameters of the mutations are assigned randomly under their constraints. After that, the input mutator determines which actions in m will be applied to b where mutating actions (b, m) are formed and test samples can be generated. The test samples will be run in DL framework (e.g., TensorFlow) whose output data is saved as expected results. The Input Data contains models and their expected results. The coverage criterion takes the mutated samples to check whether current coverage increases. If current coverage increases, the new input data will be added to test set; otherwise, such data will be discarded. This process runs until reaching a preset threshold, the target number of test samples.

Algorithm. The algorithm of our graph-based fuzz testing is shown in Algorithm 1. In procedure of FuzzWorkflow, inputs are block corpus (BC), mutations (M), a termination condition ($tc0$) that is a target number of new inputs. The while loop in Line 2 iterates until $tc0$ is reached. In Line 3, blocks are chosen by the block chooser. In Line 4, the mutation selector selects mutations and their parameters. In Line 5, Input Mutation generates test set I_k by the blocks and mutations. In Line 6, the operator-level coverage of I_k is calculated. In Line 7, $coverage_k$ is checked whether it produces additional coverage. In Line 8, MCTS Simulation is made. In Line 9, current test set, results and current operator-level coverage are updated. In Line 10, MCTS Back propagation updates back the result of the inference to update values associated with the nodes on the path from child node C to root node R.

In procedure of InputMutation, inputs are selected blocks b_k, selected model-level mutations $model\_m_k$ and source-level mutations $source\_m_k$. In Line 16, graphs are generated

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Algorithmic description of our fuzz testing framework

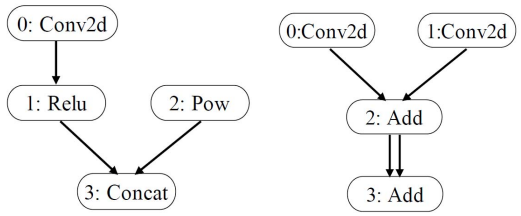
```
1: procedure FuzzWorkflow(BC, M, tc0)
2:     while not tc0 do
3:         b_k, C = BlockChooser(R, tc1, tc2, coverage)
4:         source_m_k, model_m_k = MutationSelector (b_k, M)
5:         I_k = InputMutation (b_k, source_m_k, model_m_k)
6:         coverage_k = CoverageCriterion (I_k)
7:         if IsNewCoverage (coverage_k) then
8:             result_k = MCTSSimulation (I_k)
9:             update result, coverage, I
10:            MCTSBackpropagation (C, R, result, coverage)
11:        end if
12:    end while
13:    return result, coverage, I
14: end procedure
15: procedure InputMutation(b_k, source_m_k, model_m_k)
16:    g = GenerateGraph (b_k, model_m_k)
17:    s, p = CalcShapesParameter (g, source_m_k)
18:    I_k = GenerateModel (g, s, p)
19:    return I_k
20: end procedure
21: procedure BlockChooser(R, tc1, tc2, coverage)
22:    L = MCTSSelection (R, tc1)
23:    C = MCTSExpansion (L, coverage, tc2)
24:    b_k = GetNodesFromPath (C)
25:    return b_k, C
26: end procedure
```

from the selected blocks b_k and model mutations. In Line 17, input shapes and parameters of each block are generated from the graphs and data mutations. In Line 18, according to the graphs and parameters, the test set I_k is generated.

In the procedure of BlockChooser, inputs are the root node R (no operator is set for the root node) of MCTS tree, termination condition $tc1$ is the maximum levels of the search tree the MCTS can go down, termination condition $tc2$ is the maximum times a node can be explored. In Line 21, choose blocks for InputMutation. In Line 22, MCTS Selection is made. A leaf node L is returned. In Line 23, MCTS Expansion is applied to create a new child node C of the leaf node L. The child node C could be the lowest coverage operator or a subgraph containing it, and is not chosen in the path before. In Line 24-25, the index of C and blocks along the path from child node C to root node R are returned.

D. Block Corpus

The fuzz testing process maintains a block corpus containing blocks and their attributes, including block name, allowed range of in-degree and out-degree, inner edges of the block. When a block is a subgraph, its block name is defined as the sequence of operators in the subgraph (i.e., block Conv2d+Relu+Pow+Concat in Fig. 5(a)) and its inner edges are required. Each element in the adjacency list of inner edges is a pair of source and destination operator index. Taking an operator Conv2d and two subgraphs (shown in Fig. 5) for example, Conv2d has exactly one input, and the two subgraphs have two respectively. Allowed range of out-degree of the two are set by test framework, such as {0,1,2}. Inner edges of the two subgraphs are {(0, 1), (1, 3), (2, 3)} and {(0, 2), (1, 2), (2, 3) , (2, 3)} respectively.



(a) Conv2d+Relu+Pow+Concat (b) Conv2d+Conv2d+Add+Add

Fig. 5. Block structures of Conv2d+Relu+Pow+Concat and Conv2d+Conv2d+Add+Add

E. Block Chooser

The block chooser is designed for DL models generation to improve operator-level coverage and suppress duplicated exceptions. In tests that randomly select operators to build DL models, a large number of duplicate exceptions are found. In block chooser, Monte Carlo Tree Search (MCTS) is used to search the input domain of DL inference engines, such that the most promising blocks can be chosen in block chooser to generate stochastic DL models. Each node of the tree represents an operator in the block corpus. MCTS dynamically adapts itself to the most promising search regions, where good combinations are likely to find more exceptions. MCTS process shown in Fig. 6 can be divided into the following four steps.

Selection: Starting from the root node R, successively select child nodes according to their potentials until a leaf node L is reached. The potential of each child node is calculated by using UCT (Upper Confidence Bound applied to Trees) [19] [20]. UCT is defined as:

$$potential = \frac{v}{n} + e \times \sqrt{\frac{lnN}{n}} \tag{8}$$

where v refers to the success count of the node, n is the visit count of the node, and N is the visit count for the parent of the node. e is a hyper parameter determining exploration-exploitation trade-off. The maximum levels of the search tree that the MCTS can go down is set as terminal condition 1 ($tc1$).

Expansion: Unless L is a terminal node with the highest potential, create one child node (operator) C with the lowest coverage and the operator C is not in the path. We pick a operators or a subgraph that contains the operator C.

Simulation: Generate stochastic DL models using the blocks in the current path of tree until reaching a terminal condition, and then inference the models. The maximum times a MCTS node can be explored is set as terminal condition 2 ($tc2$).

Back propagation: Propagates back the result of the inference to update values associated with the nodes on the path from C to R. The path containing the nodes with the highest

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

values in each layer would be the optimal strategy in the test set.

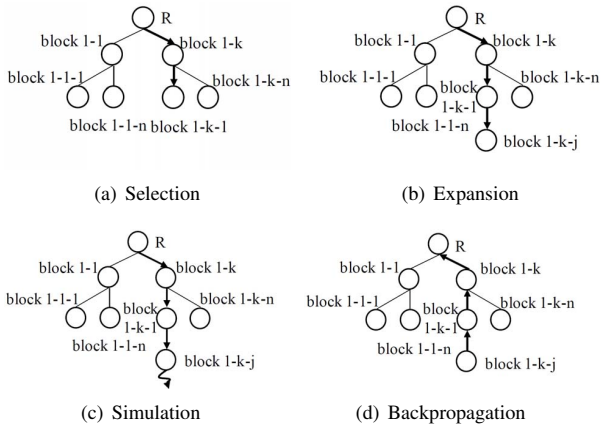


Fig. 6. Scheme of a Monte-Carlo Tree Search

F. Mutatation Selector

Mutations. The stochastic graphs generated by graph models above, only cover a small set of n-node graphs. Mutations can extend the graphs for high coverage. The framework applies these mutations by selecting different predefined mutation parameters, including 4 model-level mutations and 2 source code-level mutations.

Model-level mutations are applied to the initial digraph and blocks. Let $E(g)$ be the node count of graph g. Let r ($0 \leq r < 1$) be the probability of model-level mutations.

- **Graph Edges Addition (GEA)**. Add $\lceil E(g) \cdot r \rceil$ edge to graph g.
- **Graph Edges Removal (GER)**. Delete $\lfloor E(g) \cdot r \rfloor$ edge from graph g.
- **Block Nodes Addition (BNA)**. Duplicate an operator to every subgraph of graph g with probability r.
- **Block Nodes Removal (BNR)**. Remove an operator and its edges from every subgraph of graph g with probability r.

Two source-level mutations mutate input shape of the network and operator parameters after blocks selected for nodes in digraph.

- **Tensor Shape Mutation (TSM)**. Mutate the shape of the input tensor.
- **Parameters Mutation (PM)**. Variation of input parameter. Selecting a random enumeration for a discrete type and a random value within range for continuous type.

G. Input Mutator

To generate a DL model, the following steps are applied. First, network generator generates a digraph with a specific graph model and update connections of the graph by the model mutation methods. For each node in the graph, a block with the same input degree is selected from the block corpus. Second, shapes&parameters calculator calculates input shape and parameters for each input. Finally, generate models and test samples for running.

Network Generator. Two random graph models in graph theory are applied. One is Watts-Strogatz (WS) model proposed by Watts et.al. [21], and the other is Residual Network (RN) model we proposed in this paper. The RN model generates residual blocks in models. Let n be the node count. Let k ($k \geq 2$) be the maximum neighbors. Initially, add the n nodes $i = 0, ..., N-1$ sequentially in a line. Let $k\_current_i$ ($1 \leq k\_current_i \leq k$) be the neighbor count of node i. For every node whose current neighbor count is less than k, Add edges connecting a node i to another node j ($i < j$ and $k\_current_j < k$) with probability p ($0 < p \leq 1$), and repeat this step $k - k\_current_i$ times for node i. k, p and n are the parameters of the RN model, denoted as $RN(k, p, n)$.

Shapes&parameters Calculator. Shapes&parameters calculator involves satisfying the demands of structuring DL neural network. Those shape-free parameters are randomly selected from their range. Two methods are used to focus on the effectiveness of DL models with various input shapes.

- **Aggregation**,including Add, Concat, etc. We use operators such as Cast, Shape, Slice, and Pad to convert the mismatched input shape or data type into expected form before these aggregations. Pad is merged into adjacent operators, such as Pooling, Conv2d, DepthwiseConv2d, Pad, etc.
- **Operators with padding**, including Pooling, Conv2d, DepthwiseConv2d, etc. The padding of these operators are calculated to keep the shapes of input and output consistent. Taking Conv2d with 'SAME' padding mode for example, given input shape [iN,iH,iW,iC] and other parameters, the output height oH is computed as (9), where pH is padding height, fH is filter height, sH is stride height, and dH is dilation height.

$$oH = (iH + 2 \cdot pH - dH \cdot (fH - 1))/sH \quad (9)$$

Regarding the shapes of layer input and output are consistent, the following can be obtained:

$$oH = iH \quad (10)$$

Then other parameters are associated, satisfying three conditions of Conv2d as below, where Max_sH is maximum of stride height, Max_dH is maximum of dilation height, and fH is maximum of pH.

$$0 \leqslant pH \leqslant fH \quad (11)$$

$$1 \leqslant sH \leqslant Max\_sH \quad (12)$$

$$1 \leqslant dH \leqslant Max\_dH \quad (13)$$

Similarly, parameters of filters can be computed in the same way. Thus with given input shape, parameters of input vector [iN, iC, iH, iW, fN, fC, fH, fW, pad, stride, dilation] that satisfy (9)-(13), can be generated randomly with less Pads.

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

IV. Experiment Setup

A. Research questions

In this paper, we will answer the following six research questions.

RQ1: How effective is the approach in detecting exceptions of DL inference engine?

RQ2: How does the MCTS-based search algorithm perform comparing with random search for decision processes?

RQ3: How effective is the RN model in increasing operator-level coverage and detecting exceptions?

RQ4: How effective is the mutation strategy in increasing operator-level coverage criterion and detecting exceptions?

RQ5: How effective is the subgraph of the approach?

RQ6: Are these exceptions found related to the operator-level coverage?

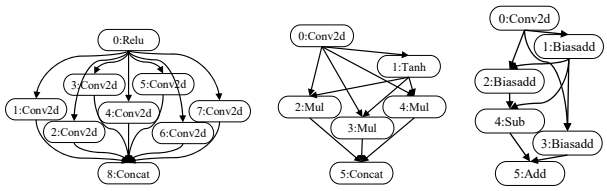
B. Experiment setup

We set up our experiments as follows.

Block Corpus. Block corpus in this experiment consists of 53 blocks, including 50 operators and 3 subgraphs.

(1) 50 TensorFlow operators supported by MNN in reference guide [22]: Add, Argmax, Avgpooling, Batchtospacend, Biasadd, Cast, Ceil, Concat, Conv2d, Cropandresize, Deconv2d, Depthwise, Exp, Expanddims, Fill, Fusedbatchnorm, GatherV2, Greater, Lrn, Maximum, Maxpooling, Minimum, Mul, Realdiv, Reducemax, Reducemean, Reduceprod, Reducesum, Relu, Relu6, Reshape, Resizebilinear, Resizenearestneighbor, Rsqrt, Selu, Shape, Sigmoid, Resize, Slice, Softmax, Spacetobatchnd, Sqrt, Square, Squeeze, Stridedslice, Sub, Tanh, Tile, TopKV2, Transpose.

(2) Subgraphs. Subgraph 1 (Fig. 7(a)) refers to the concatenation of multiple feature maps in SSD [23]. Subgraph 2 (Fig. 7(b)) is inspired by operator fusion and arithmetic optimizer in TensorFlow graph optimization. Subgraph 3 (Fig. 7(c)) is inspired by operator fusion in MNN. We set the range of output degree as $\{0, 1, 2, 3, 4, 5\}$.



(a) Subgraph 1. (b) Subgraph 2. (c) Subgraph 3.

Fig. 7. Three subgraphs in block corpus

Inference Runtime. Our experiments are conducted with MNN (V1.0.2) on X86 CPU (2.10GHz $\times$ 4) and ARM CPU (Honor X10). Both inference runtimes are calculated using FP32. Test inputs and their inference results are generated by TensorFlow (V1.12, CPU mode). Inputs, filters and biases are generated from the uniform distribution [-1,1]. The final status of the test process includes model conversion failure (MCF), inference failure (IF), data comparison failure (DCF) and data comparison pass (DCP). The comparison threshold between MNN and TensorFlow is that the ratio of data with a relative error greater than 0.1% to the total data is less than 0.1%.

Formally, let $RE(mnn)$ be the ratio of the numbers with the relative error between MNN and TensorFlow over the total output data of an operator. When $RE(mnn) \geq 99.9\%$, the result is considered as a success.

Some failures are caused by the same defect. In order to eliminate duplication, model conversion failures with the same error type, error code and failure operator are regarded as the same error, and data comparison failures with the same structure and operator are regarded as duplicates.

Configuration. In order to generate diversified models, the parameters are set in a wide range. The probability of four models-level mutations is chosen from $\{0, 0.1, 0.2\}$. In graph algorithm, the number of neighbors k is chosen from $\{2, 4, 6\}$. The probability of rewriting connections p of WS is 0.5. The probability p of RN is 0.9. For operator-level coverage, the n_{maxspc} of Formula(5) is set to 200. The weights of Formula (6) and (7) are set to 1. In MCTS-based block chooser, $tc1$ is set to 10, $tc2$ is set to 1 and e is set to $1/\sqrt{2}$. Each terminal node can expand up to 3 child nodes.

For RQ2, RQ3 and RQ4, 400 test inputs are generated for each strategy, and they are inferred on x86 CPU only, because MNN cannot infer models with multiple inputs or multiple outputs on ARM CPU. The block number of models for each strategy is set to 5, 10 and 15 respectively. For RQ1 and RQ5, the generated models are inferred on X86 CPU,and the single-input and single-output models of these models are also inferred on the ARM CPU.

V. Evaluation

A. RQ1: How effective is the approach in detecting exceptions of DL inference engine?

In order to answer RQ1, we generated approximately 1000 models and found more than 40 different exceptions. The number of blocks for models is chosen uniformly from 1 to 30. Some typical exceptions are as below.

Models Conversion Failure (MCF).

MCF-1 Segmentation fault in TopKV2 conversion. MNN model cannot be generated. MNN Log: Error for 18. Segmentation fault (core dumped). Almost all models that include TopKV2 operators cannot be converted and inferred successfully.

MCF-2 Conversion aborted. The deconv API of the pb model(shown in Fig. 8) is tf.nn.conv2d_transpose. MNN log: /converter/source/common/writeFb.cpp:108: Check failed: (notSupportOps.size()) == (0). Not Support: Tensorflow::Conv2DBackpropInput, output_shape is not consistent with inferred output shape in MNN. (height, width): (38,102) vs (4,21). Convert Tensorflow's Op deconv2d_outputdata_10100, type = Conv2DBackpropInput, failed. Aborted (core dumped). It reveals that the shape of the output tensor calculated by the operator Deconv in MNN is wrong.

Inference Failure(IF). **IF-1 Inference aborted**. The Reduceprod cannot output results (shown in Fig.

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

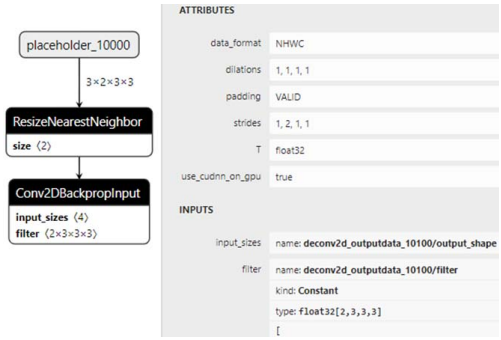


Fig. 8. MCF-2 Conversion aborted: pb model structure and deconv's parameters.

9). MNN Log: Error in 'python': free(): invalid next size (fast): 0x000000000 1e2cb90. Backtrace: .../dist-packages/_mnncengine.so (_ZN3MNN6TensorD1Ev+0x74) [0x7f7672028654]. Aborted (core dumped).

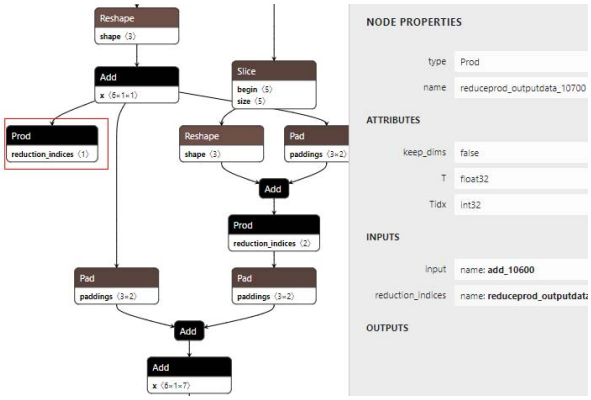


Fig. 9. IF-1 Inference aborted: pb model structure and Reduceprod's parameters. Inference aborted: the reduceprod cannot output result.

Data Comparison Failure(DCF) DCF-1 core dumped and data comparison failure in Sub. The model structure is shown in Fig. 10. $RE(mnn_{X86CPU})$ of Sub1 is 52.08%, $RE(mnn_{X86CPU})$ of Sub2 is 76.04%. MNN log: Error in 'python': double free or corruption (!prev): 0x0000000002 4ae8b0. Backtrace: /dist-packages/_mnncengine.so (_ZN3MNN15BufferAllocator4NodeD1Ev+0x88) [0x7ff0e81 7a098]. Aborted (core dumped).

DCF-2 Data Comparison Failure of operators. (1) When converting some FP32 numbers(from 0 to 1) into INT8 numbers, the cast operator will yield a calculation error. (2) When the input of a sigmoid operator is nan (coming from Rsqrt), the results are nan in TensorFlow and 1 in MNN respectively. (3) In Fig. 11(a), $RE(mnn_{ARMCPU})$ of Avgpooling is 33.33%. (4) In Fig. 11(b), $RE(mnn_{ARMCPU})$ of Maxpooling is 50.00%. (5) In Fig. 11(c), $RE(mnn_{ARMCPU})$ of Relu is 45.19%. The Conv2d of the model (Fig. 11(c)) caused an incorrect calculation result. (6) The result of Realdiv (Fig. 11(d)) divided by zero is inf in TensorFlow and nan in MNN

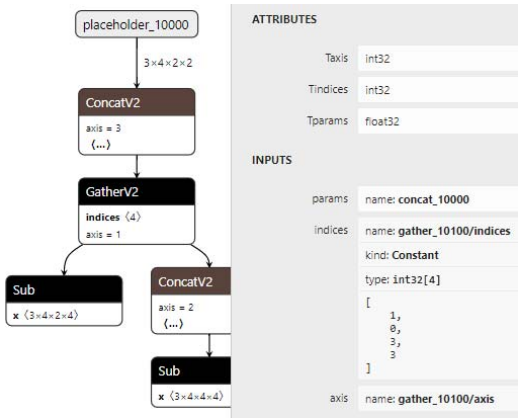
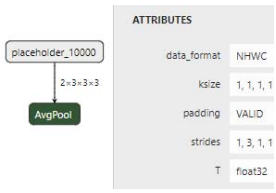
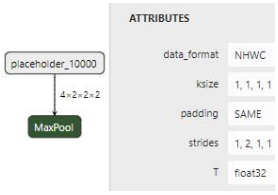


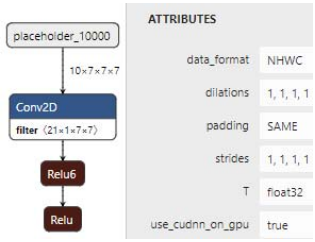
Fig. 10. DCF-1: pb model structure and Gather's parameters. Data Comparison Failure of Sub.



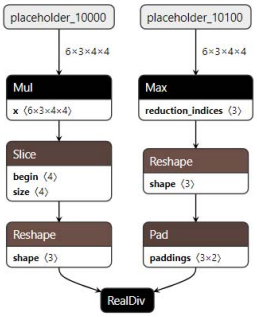
(a) $RE(mnn_{ARMCPU})$ of Avgpooling is 33.33%.



(b) $RE(mnn_{ARMCPU})$ of Maxpooling is 50.00%.



(c) $RE(mnn_{ARMCPU})$ of Relu is 45.19%.



(d) $RE(mnn_{ARMCPU})$ of Realdiv is 91.67%.

Fig. 11. DCF-2 Data Comparison Failure of single operators

X86 CPU.

Model Generation Failure. It is worthy of mentioning that some exceptions of TensorFlow are found in model generation. When generating a model containing two Concats whose two inputs come from the same two constants, TensorFlow will get stuck.

Answer to RQ1: Our approach is effective to detect various exceptions in model conversion and inference process for the DL inference engine, such as crashes, inconsistencies, nan and inf bugs.

B. RQ2: How does the MCTS-based search algorithm perform comparing with random search for decision processes?

In order to answer RQ2, we evaluate operator-level coverage and inference results using MCTS-based search and random

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

search for block chooser.

Operator-level coverage. As shown in TABLE III (1)(5), the operator-level coverage of random search and MCTS-based search are 60.2% and 61.5% for 5 blocks, 69.9% and 76.4% for 10 blocks, 71.8% and 81.7% for 15 blocks respectively. MCTS-based search, on average, covers 6.7% more operator-level coverage than random search.

Inference results. We measure the exceptions of inference results for each search algorithm. TABLE III(1)(5) shows the exceptions found. MCTS-based search, on average, finds 9.7 more exceptions than random search. It can be observed that the average unduplicated rate of random search is 11.6%, which is lower than that of MCTS search (28.3%). Using the MCTS-based search takes an average of 1.1 hours longer than using the random search. Because the efficiency of MCTS-based search that find exceptions is significant, a small increase in execution time is acceptable for industrial testing.

Answer to RQ2: The MCTS-based block chooser outperforms the random-based block chooser in boosting operator-level coverage (6.7% more) and detecting exceptions (9.7 more).

C. RQ3: How effective is the RN model in increasing operator-level coverage and detecting exceptions?

In order to answer RQ3, operator-level coverage and inference results are evaluated using two stochastic network generation strategies: (1) WS and RN model together, each model generating half of the test samples. We do not aim to verify that whether a graph model is superior to other models in certain configurations. The RN model is only used to generate more diversified models. (2) WS model only. (3) RN model only. Five input shapes can be chosen uniformly for each model. To avoid disturbing the coverage and inference result of random graph models, mutations are cancelled.

Operator-level coverage. As shown in TABLE III(2)(3)(4), there are two key observations from the results. First, WS and RN model together, on average, covers 3.6% and 1.47% more operator-level coverage than WS model only and RN model only respectively as demonstrated. Second, the coverage rate when n is 15 is slightly higher than when n is 10. This is intuitive as a higher value (N = 15) of blocks making it increasingly harder to cover more topologies and types of shapes&parameters without mutations.

Inference results. The exceptions of inference results are measured for each strategy. TABLE III(2)(3)(4) shows the detailed results. WS and RN model together, on average, finds 5.9 and 1.5 more exceptions than WS model only and RN model only respectively as demonstrated.

Therefore WS and RN model together is more efficient in finding exceptions as well as increasing blocks of DL models. The experiment also shown that the performance of RN is slightly better than that of WS.

Answer to RQ3: Different graph models can be used to generate more diverse models. Through applying the RN model to stochastic network generation strategy, more exceptions for each strategy can be detected as well as operator-level coverage increased.

D. RQ4: How effective is the mutation strategy in increasing operator-level coverage criterion and detecting exceptions?

In order to answer RQ4, we evaluate operator-level coverage and inference results of test generation with mutations and test generation without mutations. For test generation without mutations, 5 input shapes can be chosen uniformly for each model without any mutations.

Operator-level coverage. The operator-level coverage of two strategies are 59.2% and 61.5% for 5 blocks, 65.8% and 76.4% for 10 blocks, 69.8% and 81.7% for 15 blocks respectively. For test generation with mutations, on average, covers 8.2% more operator-level coverage than test generation without mutations as demonstrated in TABLE III(4)(5).

Inference results. The exceptions of inference results are measured for each strategy. TABLE III(4)(5) shows the detailed results. The mean number of exceptions are 16.7 and 20.1 for 5 blocks, 20.2 and 31.9 for 10 blocks, 19.5 and 31.2 for 15 blocks respectively. It can be observed that test generation with mutations is more efficient in finding exceptions(on average, 8.6 more exceptions) as well as increasing operator-level coverage of inputs.

Answer to RQ4: Our mutation strategy is useful to generate new valid test inputs, by up to 8.2% more operator-level coverage and 8.6 more exceptions detected on average. It can be observed that our mutation strategy is effective for generating diversified models.

E. RQ5: How effective is the subgraph of the approach?

In order to answer RQ5, 100 mutated subgraphs (MS) are generated to evaluate the effectiveness of three subgraphs in block corpus (shown in Fig. 7). The number of blocks in a model is chosen from $\{1, 3\}$. The success ratios are 45.2% on ARM CPU and 100% on X86 CPU. The main type of error is data comparison failure on ARM CPU. Inference exceptions are analyzed as below. Fig. 12 shows two typical exceptions of mutated subgraphs.

MS-1. Data Comparison Failure. This mutant deletes a Mul from subgraph 2 (Fig. 12(a)) and $RE(mnn_{ARMCPU})$ of Relu is 33.33%.

MS-2. Data Comparison Failure. This mutant a Biasadd from subgraph 3, and delete a Conv2d from subgraph 1 (Fig. 12(b)) and $RE(mnn_{ARMCPU})$ of Concat is 8.93%. when given more than 2 inputs, it reveals that the operator Concat will lose some of the input data and outputs wrong results.

Answer to RQ5: It is difficult to generate a model of a specific structure only by matching operators with nodes in a random graph. Subgraphs and their mutants can be helpful to construct these specific structures with more exceptions detected.

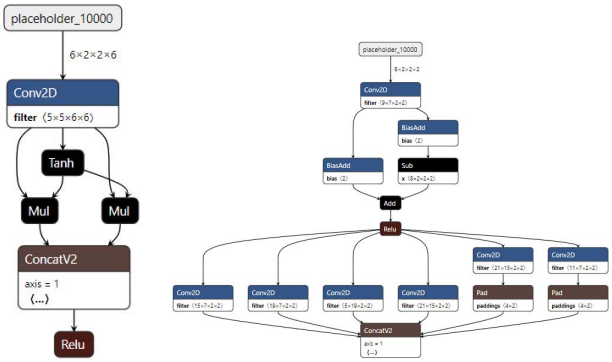
F. RQ6: Are these exceptions found related to the operator-level coverage?

In order to answer RQ6, relations between the typical exceptions and operator-level coverage are analyzed.

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

TABLE III
OPERATOR-LEVEL COVERAGE, EXCEPTIONS FOUND (AFTER DEDUPLICATION/TOTAL) AND DURATION UNDER DIFFERENT BLOCK NUMBER (N = 5, 10, 15) OF MODELS FOR RQ2, RQ3 AND RQ4.

				N = 5			N = 10			N = 15		
	Graph model	Mutations	Search	OLC	Exceptions found (d/t)	Duration (h)	OLC	Exceptions found (d/t)	Duration (h)	OLC	Exceptions found (d/t)	Duration(h)
1	WS and RN	Yes	Random	60.2%	12/103	1.6	69.9%	20.1/161	2.7	71.8 %	21.7/201	3.4
2	WS	NO	MCTS-based	57.2%	10.9/75.3	2.7	62.9%	13.7/95.4	4.1	64.8%	14.2/91.8	4.4
3	RN	NO	MCTS-based	57.7%	14.4/77.2	2.6	65.3%	19.5/92.8	4.1	67.3%	17.9/113.4	4.5
4	WS and RN	NO	MCTS-based	59.2%	16.7/81.2	2.8	65.8%	20.2/93.1	4.1	69.7%	19.5/118.9	4.5
5	WS and RN	Yes	MCTS-based	61.5%	20.1/84.5	2.7	76.4%	31.9/89.6	4.1	81.7%	31.2/120.3	4.5



(a) Delete a Mul from subgraph 2. (b) Delete a Biasadd from subgraph 3, and delete a Conv2d from subgraph 1.

Fig. 12. Inference results of 3 mutated subgraphs in block corpus.

Operator Type Coverage. We note that some operators are supported in MNN reference guide, but the error reported is that this operator is not supported in model conversation, such as Deconv in MCF-2. Almost all models that include TopKV2 operators cannot be converted and inferred successfully. Our result(MCF-1) confirm that the TopKV2 operator is almost unsupported in current MNN version. We also tried to infer some models that contain operators unsupported in MNN reference guide, such as Addn, Clip and Asin. These models are successfully inferred by MNN on X86 CPU and ARM CPU.

Single Edge Coverage. Note that SEC is usually associated with **Input Degree Coverage** or **Output Degree Coverage**, such as multi-output (e.g., add of IF-1) and multi-input operators (e.g., (6) Realdiv of DCF-2 and Concat of MS-2). Some structures also cause special inputs for operators, such as nan of (2) Rsqrt and Sigmoid in DCF-2.

Shapes&Parameters Coverage. Some results of operators with specific value of parameters or tensor shapes are unexpected in comparison with TensorFlow, such as Avgpooling and Maxpooling in DCF-2.

Answer to RQ6: In summary, the exceptions detected are all within the scope of operator-level coverage. In addition, there is no obvious correlation between each metrics of operator-level coverage and a certain error types. That is, guided by these metrics, the inputs can trigger various types of exceptions.

VI. THREATS TO VALIDITY

The internal threat to validity mainly lies in the implementations of our method and scripts in the experiment. All the artifacts are maintained by experienced software engineers in Huawei. Besides, the method has been used in Huawei company, which reduces the internal threats.

The external threats to validity mainly lie in the library and DL inference engine used in the experiment. To reduce this threat, we adopted one of the most widely-used library TensorFlow to generate input DL models. Then, we chose the Alibaba MNN as the target DL inference engine under test (X86 CPU and ARM CPU). Furthermore, the proposed method tests a DL inference engine of Huawei for several months with many valid exceptions detected.

The construct threats to validity mainly lie in settings, randomness and measurements in our experiment. (1) To reduce the impact of settings, we constructed a block corpus (50 operators and 3 subgraphs) and five experiments. (2) To reduce the impact of randomness, a large number of models were generated in every experiment respectively (i.e., 1000 for RQ1, 400 for RQ2-RQ4). We repeated each method for 10 times and calculated the average results in RQ2-RQ4. (3) To reduce the impact of measurements, we carefully set a threshold $RE(mnn)$ to check whether the inference results are correct. When counting the number of exceptions, we identified the duplicated ones. Further, we manually generated DL models and triggered all the detected exceptions successfully.

VII. RELATED WORK

Fuzz Testing and Mutation Testing. Fuzz testing is a widely used technique for exposing defects in DL system. Guo et al. [24] proposed the first differential fuzz testing framework for DL systems. TensorFuzz proposed by Odena et al. [10], used a nearest neighbour hill climbing approach to explore achievable coverage over valid input space for TensorFlow graphs, and to discover numerical errors, disagreements between DL models and their quantized versions. Pei et al. presented DeepXplore [11] which proposed a white-box differential testing technique to generate test inputs for DL system. Wicker et al. [25] proposed feature-guided test generation. They transformed the problem of finding adversarial examples into a two-player turn-based stochastic game. Ma et al. [12] proposed Deepmutation which mutates DNNs at the source level or model level to make minor perturbation on the decision boundary of a DNN. Shen et al. [26] proposed

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

five mutation operators for DNNs and evaluated properties of mutation . Xie et al. [27] presented a metamorphic transformation based coverage guided fuzzing technique, DeepHunter, which leverages both neuron coverage and coverage criteria presented by DeepGauge [8]. Existing testing techniques focus on the quality of DL models but lacks attention to the core underlying inference engines (i.e., frameworks and libraries). Our method generates models as the input of the fuzz testing for DL inference engine. Together with the combinations of operators, we design new mutation rules to generate diversified DL models to trigger different structured parts of a given DL inference engine.

Test Coverage Criteria. Coverage criteria is an important part of testing methods. Code coverage is most popular for conventional software testing, but it does not make sense for DL testing, since the decision logic of a DL model is not written manually but rather it is learned from training data [28]. In the study [11], 100 % traditional code coverage is easily achieved by a single randomly chosen input. Pei et al. [11] proposed the first Coverage criterion: neuron Coverage. Neuron coverage is calculated as the ratio of the number of unique neurons activated by all test inputs and the total number of neurons. Ma et al. [15] proposed layer-level Coverage, which considers the top hyperactive neurons and their combinations to characterise the behaviours of a DNN. Du et al. [26] first proposed State-level Coverage to capture the dynamic state transition behaviours of deep neural network. Li et al. [29] pointed out the limitations of structural coverage criteria for deep networks caused by the fundamental differences between neural networks and human-written programs. DeepCover [30] proposes the test criterion [31] for DNNs, adapted from the MC/DC test criterion of traditional software. Existing neural coverage criteria of DL models cannot work in DL inference engine testing scenario, because the inputs for testing DL inference engines are DL models. Thus, a novel criterion is required to capture behaviors of DL inference engines rather than those of DL models. Our proposed operator-level coverage is naturally followed by the graph structure of DL models. The results show that the operator-level coverage guided testing framework improves the effectiveness in detecting exceptions.

VIII. Conclusion

The issues triggered by a single specific model are limited and diversified combinations of operators (models) are more capable of triggering DL inference engine issues. To overcome the challenge of generating such models with diversified combinations of operators, this paper employs graph-based fuzz testing incorporating six different mutations, MCTS and a novel operator-level coverage criterion proposed based on graph theory. One possible application scenario is implemented on MNN, and the results demonstrate the effectiveness of our proposed method. In fact, the proposed method has been used continuously in a DL inference engine of Huawei for several months, which finds many valid exceptions during the internal build and is efficient for industry.

Acknowledgement

We would like to thank anonymous reviewers for insightful comments; we also thank Jiawei Liu for discussions on the manuscript. This work is supported partially by National Natural Science Foundation of China (61932012, 61802171), and Fundamental Research Funds for the Central Universities (14380021).

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[2] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, "Achieving human parity in conversational speech recognition," *CoRR*, vol. abs/1610.05256, 2016. [Online]. Available: http://arxiv.org/abs/1610.05256

[3] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang *et al.*, "Mlperf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[4] "Nvidia tensorrt," 2018. [Online]. Available: https://developer.nvidia.com/tensorrt

[5] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *arXiv preprint arXiv:1907.01989*, 2019.

[6] "Alibaba mnn," 2020. [Online]. Available: https://github.com/alibaba/MNN

[7] 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*, vol. 41, no. 5, pp. 507–525, 2014.

[8] X. Xie, L. Ma, X. Juefei, C. Felix, M. Hongxu, Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Deephunter: Hunting deep neural network defects via coverage-guided fuzzing," *arXiv preprint arXiv:1809.01266*, 2018.

[9] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[10] A. Odena and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," *arXiv preprint arXiv:1807.10875*, 2018.

[11] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[12] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei, Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *IEEE 29th International Symposium on Software Reliability Engineering*. IEEE, 2018, pp. 100–111.

[13] A. Kerr, D. Merrill, J. Demouth, and J. Tran, "Cutlass: Fast linear algebra in cuda c++," 2017. [Online]. Available: https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda

[14] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 303–314.

[15] L. Ma, F. Juefei, Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[16] R. J. Trudeau, *Introduction to graph theory*. Courier Corporation, 2013.

[17] "Google. xla: Accelerated linear algebra." 2017b. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[18] R. Wei, L. Schwartz, and V. Adve, "Dlvm: A modern compiler infrastructure for deep learning systems," *arXiv preprint arXiv:1711.03016*, 2017.

[19] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*. Springer, 2006, pp. 282–293.

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.

[20] G. M. J. Chaslot, M. H. Winands, H. J. V. D. HERIK, J. W. Uiterwijk, and B. Bouzy, "Progressive strategies for monte-carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 03, pp. 343–357, 2008.
[21] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, p. 440, 1998.
[22] "Mnn supported ops," 2019. [Online]. Available: https://www.yuque.com/mnn/en/ops
[23] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European Conference on Computer Vision*. Springer, 2016, pp. 21–37.
[24] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.
[25] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 408–426.
[26] W. Shen, J. Wan, and Z. Chen, "Munn: Mutation analysis of neural networks," in *IEEE International Conference on Software Quality, Reliability and Security Companion*. IEEE, 2018, pp. 108–115.
[27] X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Coverage-guided fuzzing for deep neural networks," *arXiv preprint arXiv:1809.01266*, vol. 3, 2018.
[28] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.
[29] Z. Li, X. Ma, C. Xu, and C. Cao, "Structural coverage criteria for neural networks could be misleading," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE, 2019, pp. 89–92.
[30] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.
[31] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

Authorized licensed use limited to: Nanjing University. Downloaded on January 19,2023 at 13:08:37 UTC from IEEE Xplore. Restrictions apply.