Graph-Based Fuzz Testing for Deep Learning Inference Engine

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
Email: fangchunrong@nju.edu.cn

***Abstract*—Testing deep learning (DL) systems are increasingly crucial as the increasing usage of DL system in various domains. Existing testing techniques focus on testing the quality of specific DL models, but lacks attention to the core underlying inference engines(frameworks and libraries) on which all DL models depend. In this study, we designed a novel graph-based fuzz testing framework to test DL inference engine. The proposed framework adopts a operator-level coverage based on graph theory and implements six different mutations to generate diverse DL models by exploring combinations of model structures, parameters and data. It also employs Monte Carlo Tree Search to drive the decision processes of DL model generation. The experimental results show that: (1) guided by operator-level coverage, our approach is effective in detecting three types of undesired behaviors: models conversion failure, inference failure, output comparison failure; (2) the MCTS-based search outperforms random-based search in boosting operator-level coverage and detecting exceptions; (3) our mutants are useful to generate new valid test inputs with original graph or sub graph, by up to a 21.6% more operator-level coverage on average and 24.3 more exceptions captured.**

***Index Terms*—deep learning inference engine, graph theory, stochastic neural networks**

I. Introduction

Deep Learning (DL) is the most popular technology for dealing with many hard computational problems such as image classification, natural language processing, and speech recognition. Deep learning on edge devices, especially mobile devices, attracts growing attention. There are many advantages for deep learning on mobiles, for example, low latency, privacy protection, and personalized service. To make full use of on-device deep learning technology, inference engines tailored to mobile devices have been developed and extensively used in mobile applications, for example, Snapdragon Neural Processing Engine(SNPE) [1], HiAI [2], TensorFlow-Lite [3], MNN [4], etc. As core underlying component of a DL system, DL inference engine is invoked by applications of various fields, and enables various trained models to run on devices(such as CPUs, GPUs, DSPs, NPUs, etc) with inference acceleration. Therefore, DL inference engine, just like traditional software, must be tested systematically for different corner cases to detect and fix ideally any potential flaws or undesired behaviors, such as models conversion failure, inference failure, output comparison failure.

Fuzz testing is a widely used automated testing technique that generates random data as program inputs to detect crashes, memory leaks, failed (built-in) assertions in software. It have been proved to be effective in DL system testing. A reasonable approach [5] [6] involves randomly searching around a given input for changes that cause misclassification for trained neural networks. Other approaches [7] are to design mutation operators like add or remove a layer which requires the shape of layer input and output to be consistent(e.g. activation function), to inject various faults into the training data(e.g. weights) or given DNNs. Then the training process is re-executed to generate the mutated DL models. However, these methods are designed for testing models, and suffer from certain limitations that focus on testing a limited number of specific DNNs. For DL inference engine as underlying platform in DL systems, these methods above are difficult to provide plentiful DNNs that consist of various operators and topologies. Consequently, existing testing methods cannot meet adequately test input requirement of DL inference engine before deployment. This presents a new systems problem as automated and systematic testing of DL inference engine with thousands of neuron networks and their large-scale of parameters for all corner cases is extremely challenging and laborious.

In this work, we introduce a novel graph-based fuzzer, for achieving high coverage of all potential criteria (DNNs) in DL inference engine fuzz testing. Ultimately, our goal is to help developers to extend their test sets with new inputs so that more cases are covered. We propose a stochastic neural network generator, inspired by random graph generator [8] in Neural Architecture Search(NAS) studies, and a set of mutations for generation processes in achieving this goal. We also propose a coverage criterion, operator-level coverage, and a search algorithm using MCTS for decision processes of our method. Given an input, it aims to pick a series of blocks(operators and some specified paths of neural networks) that would result in detecting more exceptions. Contributions of this work are as follows:

- We introduce a novel graph-based fuzzing technique for testing DL inference engine, that is designed to work with operator-level coverage metrics.
- We compare MCTS-based search algorithm with random search in block chooser of our method.
- We also show that sub graphs defined in block corpus improve the effectiveness in detecting exceptions through

generating more targeted inputs with specific topologies.

II. Background

A. Workflow of DL inference engine

In a DL system, applications invoke interfaces provided by SDK of DL inference engine to load and run models. DL inference engine enables a trained model to run on devices with inference hardware accelerators using CPUs, GPUs, DSPs, NPUs, etc. Existing inference engines share a similar workflow. For instance, Snapdragon Neural Processing Engine (SNPE), as shown in Figure 1, provides a tool to convert a model trained via some DL framework to a DLC (DL Container) file, which can be loaded by the SNPE inference engine. Furthermore, the DLC file can be quantized optionally. Finally, the inference engine can load and do inference with the SNPE runtime.

One of the most important test input is the neural network model when testing an inference engine. The developer needs to confirm the capacities and constraints of an inference engine as they can ensure the neural network model will work on it. The SNPE SDK Reference Guide [9] declares the supported layers or operators of TensorFlow and Caffe [10] as well as their constraints. Sometimes an inference engine being tested is closed source, thus black-box testing is more general than white-box testing as it ignores the internal implementation.

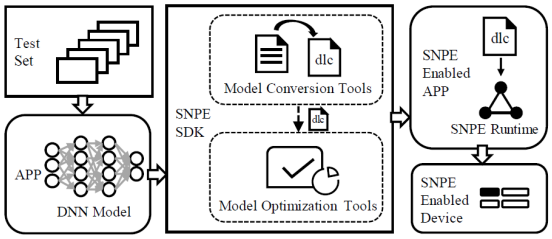


Fig. 1. Generic process of SNPE in a DL application

B. Stochastic neural network generators

Random graphs are widely studied in graph theory. Random graphs exhibit different probabilistic behaviors depending on the random process defined by the model. Xie et al. [8] propose a neural network generator to generate randomly wired graphs for networks using three classical families random graph models in graph theory: Watts-Strogatz (WS) [11], Barabasio - Albert (BA) [12] and ErdsRnyi (ER) [13] models. A random graph model (e.g. WS) only covers a small subset of all possible N-node graphs, but this subset is different from other subsets covered by other models. There are also hand-designed rules defined to map the sampled operators (e.g. Conv2d, add) to a computational DAG. The rules include, but are not limit to:

- Regular input sizes and output sizes are used in networks, analogous to stages in a ResNet. For example, 33 Conv2d, the input size is 224224 pixels, the output size is 112112 pixels.
- Sampled operators maintains the same number of output channels as input channels (unlike Concat). This prevents the Conv2d that follows from growing large in computation.
- Sampled operators are almost parameter-free, regardless of input and output degrees.

If the above rules are adopted directly in test generation for DL inference engine, the input space of allowed patterns, for example, neural network architectures, input sizes, parameters of operators, are constrained in a small subset of all possible graphs. Further, those operators whose input and output shapes are not consistent (e.g. Concat, Transpose, Pooling), are hard to be added into neural network directly, due to shape mismatch in aggregation . And the test set only covers limited testing scenarios.

C. Fuzz testing

Fuzz testing have been proved to be effective in exploring the input space of DL system testing. A coverage criterion is used to measure what percent of the potential behaviors could be tested by a given set of inputs. For this purpose, a coverage criterion divides the input space for a given system into equivalence classes and calculates how many of the equivalence classes have at least one instance input in a given set of inputs. When an equivalence class has at least one instance input in a given set of inputs, the equivalence class is said to be covered. There have been studies proposed 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. Further, DL inference engines or frameworks have made optimizations responding to demands for faster inference. CULASS [14] of CUDA provides specialized data-movement and a hierarchy of tiles to improve performance. TensorFlow [15] [16] [17] applies graph optimization to improve the performance of computations. Thereby these optimizations also challenge fuzz testing of DL inference engine to generate DNNs with various input shapes and topologies to assure correctness.

III. Methodology

In this section, we provide a detailed technical description of our methods and algorithm.

A. Definitions

Our methodology for stochastic networks generation involves the following concepts.

Digraph. Graph theory provides an excellent framework for studying and interpreting neural network topologies [18]. A neural network can be denoted by a digraph $G = (V, E)$ comprising: 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 (i.e., an edge is associated with two distinct operators). In neural networks, edges are data flow.

Sub graph. From the introduction above, some specified paths of neural network will be specially processed (e.g. graph optimization to run a faster inference). There is a very low probability that these specified paths could be generated randomly. Thus sub graphs are applied to blocks to cover those specified path directly in testing. Formally, digraph $G' = (V', E')$ is a sub graph of G iff $V' \subseteq V, E' \subseteq E \bigwedge((x, y) \in E' \rightarrow x, y \in V')$.

Block. Sub graphs or operators of a neural network are defined as blocks in this paper. And a network is constructed by operators and sub graphs as shown in Figure 2.

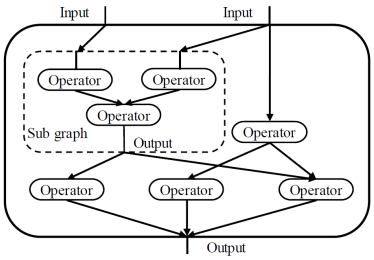


Fig. 2. A network is constructed by operators and sub graphs.

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 sub graph and can be empty otherwise.

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

As defined in III-A, we use the topological, input tensor shape, parameter to characterize behaviors of operators at operator-level. If a new test sample produces additional coverage, it will be added in input test set. Given a block corpus BC and a test set I, operator-level coverage criteria are defined as follow:

Type of Operator Coverage(TOC). Let n_t be the number of total types of operators defined in BC. Let $f_t(I)$ be the number of types of the operator in I. The TOC of I is defined as :

$$TOC(I) = \frac{f_t(I)}{n_t}$$

Input degree of Operator Coverage(IDOC). 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}: $IDOC\_op(c, I) = \frac{f_{id\_op}(I)}{n_{id\_c}}$. The IDOC of I is defined as:

$$IDOC(I) = \frac{\sum IDOC\_op(c, I)}{n_t}$$

Output degree of Operator Coverage(ODOC). 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}: $ODOC\_op(c, I) = \frac{f_{od\_op}(I)}{n_{od\_c}}$. The ODOC of I is defined as:

$$ODOC(I) = \frac{\sum ODOC(c, I)}{n_t}$$

Single Edge Coverage (SEC). Let $f_{se}(I, c)$ 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:

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

Multiple Edges Coverage(MEC). Let n_{me} be the number of different multi-input operators defined in BC. Let $MEC\_op(c, I)$ be 1 when the multi-input operator c in test set I has same inputs, and be 0 otherwise. The MEC of test set I is defined as:

$$MEC(I) = \frac{\sum MEC\_op(c, I)}{n_{me}}$$

Shapes&Parameters Coverage(SPC). Let n_{spc} be the target mean of types 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: $SPC\_op(c, I) = \frac{f_{spc}(c,I)}{n_t}$. The SPC of I is defined as:

$$SPC(I) = \frac{(\sum SPC\_op(c, I)/n_t)}{n\_spc}$$

Operator-level Coverage (OLC). Let OLC of the test set I be the weighted mean of a set of metrics $Z = \{TOC(I), IDOC(I), ODOC(I), SEC(I), MEC(I), SPC(I)\}$ with corresponding non-negative weights $\{w_1, w_2, ..., w_6\}$. Formally, OLC of I is defined as:

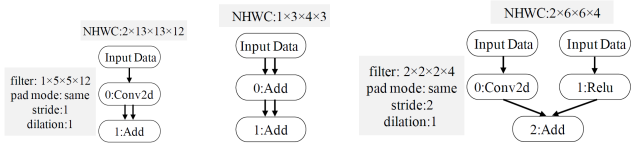
$$OLC(I) = \frac{\sum w_i m_i}{\sum w_i}, m_i \in Z, \sum w_i = 1$$

The weights sum up to 1. Some may be zero. For example, weight of $MEC(I)$ is 0 when test set I does not contain a multi-input operator, and weight of $ODOC(I)$ is 0 when expected output degree of operators are all 1 in test samples of test set I.

For example, Figure 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 . 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 S

C. Framework

The core idea of our approach is to make use of reward-guided exploration, to extend the test set by mutating graph and block so that to achieve a higher operator-level coverage score. In generating process, test inputs are modeled as a graph. In reward-guided process, operators are evaluated with their success ratio and coverage changes they induce.

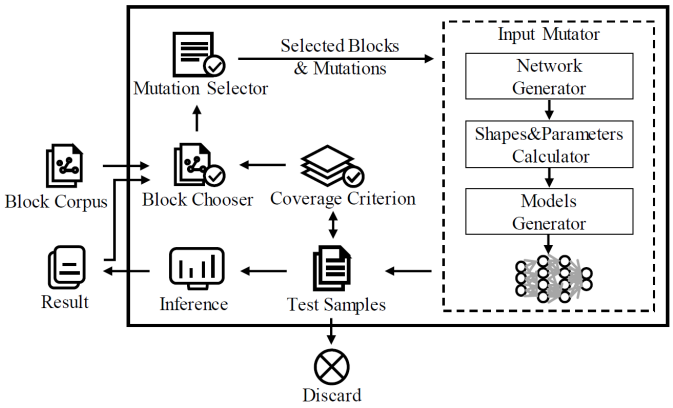


Fig. 4. Workflow for an iteration

The Framework of Fuzzer. The system 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 in that 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. And the Input Data contains models and their expected results. The coverage criterion takes the mutated inputs to check whether current coverage increase. If current coverage increases, the new input data will be added to test set, or will be discarded. This process runs until reaching the maximum of test samples. The detailed workflow is depicted as Figure 4.

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

Object	TOC	IDOC	ODOC	SEC	MEC	SPC	OLC
Conv2d	100%	100%	66.7%	33.3%	N/A	20%	64%
Relu	100%	100%	33.3%	33.3%	N/A	10%	55.4%
Add	100%	100%	33.3%	33.3%	100%	30%	66.2%
Test Set S	100%	100%	44.4%	33.3%	100%	20%	66.2%

Algorithm 1 algorithmic description of the 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
```

Algorithm. We describe our method 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 line3 refers to iterating until $tc0$. In line 3, blocks are chosen by the input selector. 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 8-9, MCTS Simulation is made. 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 C to R.

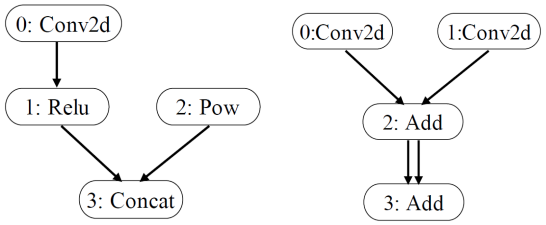
In procedure of InputMutation, inputs are selected blocks, selected model-level mutations $model\_m_k$ and source-level mutations $source\_m_k$. In line 16, graphs are generated from the selected blocks b_k and model mutations. In line 17, input shapes and parameters of each block are generated from the graphs and data mutations. In line 18, according to the graphs

and parameters, a set of models is generated as I_k .

In procedure of BlockChooser, inputs are the root node R of MCTS tree, corpus, termination condition $tc1$ that is the maximum levels of the search tree the MCTS can go down, termination condition $tc2$ that is the maximum times a MCTS node can be explored. In line 21, choose blocks for InputMutation. In line 22, MCTS Selection is made. And 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 sub graph containing it, and is not chosen in the path before. In line 24-25, index of C and blocks along the path from C to 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. The name of sub graph is defined by the sequence of operators in the sub graph(i.e. block Conv2d+Relu+Pow+Concat in Figure 5(a)). Inner edges are required when the block is a sub graph and can be empty otherwise. Each element in the adjacency list of inner edges is a pair of source and destination operator index. Take an operator Conv2d and two sub graph (shown in Figure5) for example, Conv2d has exactly one input, the two sub graph 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 sub graph are $\{(0, 1), (1, 3), (2, 3)\}$ and $\{(0, 2), (1, 2), (2, 3) , (2, 3)\}$ respectively. And Conv2d does not involve.



(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

In block chooser, we use Monte Carlo Tree Search (MCTS) to search the input domain of DL inference engine so that the most promising blocks can be chosen to generate stochastic neural network. Each node of the tree represents an operator in the block corpus. MCTS dynamically adapts itself to the most promising search regions, where good consequences are likely to follow to find more exceptions. MCTS process shown in Figure 6 can be broken down 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:

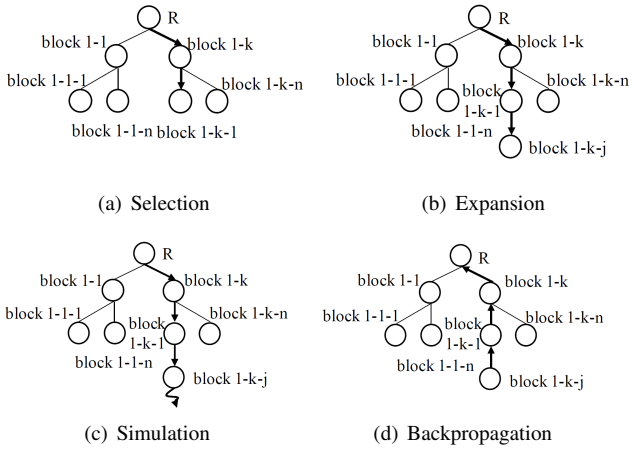
$$potential = \frac{v}{n} + e \times \sqrt{\frac{lnN}{n}} \tag{1}$$

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 the MCTS can go down is set as terminal condition 1 ($tc1$).

Expansion: Unless L is a terminal node, create one child node C. We pick the block (operators or sub graph) that contains lowest operator coverage and is not in the path as C.

Simulation: Generate stochastic neural networks 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 values in each layer would be the optimal strategy in the test set.



(a) Selection (b) Expansion

(c) Simulation (d) Backpropagation

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

F. Stochastic network generators

Input Mutator. The input mutator mutates the input according to the selected blocks and mutations. Figure 4 illustrates how the input mutation generator constructs mutated input test samples. To generate a network model, the following steps are applied. First, generate a digraph with a specific graph model and update connections of the graph by the model mutation methods. Select blocks with the same input degree from the block corpus for each node in the digraph. Second, calculate input shape and parameters for each input. Finally, generate models and test samples for running.

Network Generator. Two random graph models are applied in this paper. Watts-Strogatz (WS) [21] is a classical families

random graph model in graph theory. We propose a Model named Residual Network(RN), which adds residual block and multi-output block to networks. The RN model generates residual blocks in networks. 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 the step $k - k\_current_i$ times for node i . k, p and n are the parameters of the RN model, denoted as $RN(k, p, n)$.

Mutations. The stochastic graphs generated by graph models above, only cover a small set of n-node graphs. Mutations can extend the graphs for a more complete coverage. The input mutator applies mutations including 4 model-level mutations and 2 source-level mutations. The block set and available mutations are the hyper parameters of input mutation generator. With the hyper parameters set properly, models-level or source-level mutations can be produced. The test program enumerates these mutations and the mutation selector can identity them by their indices. 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.

- **Edges of Graph Addition (EGA)**. Add $\lceil E(g) \cdot r \rceil$ edge to graph g.
- **Edges of Graph Removal (EGR)**. Delete $\lfloor E(g) \cdot r \rfloor$ edge from graph g.
- **Nodes of Block Addition (NBA)**. Duplicate an operator to every sub graph of graph g with probability r.
- **Nodes of Block Removal (NBR)**. Remove an operator and its edges from every sub graph 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 shape of tensor shape. Take Relu for example, the shape of feature map [iN, iH, iW, iC].
- **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. Shapes&parameters Calculator

Shapes&parameters calculator involves satisfy the demands of structuring DL neural network. We use two methods to focus on the effect of network topology with various input shapes. Those shape-free parameters are randomly selected from their range.

Aggregation, including Add, Concat, etc. To keep the same input shapes, one approach is padding with zeroes before these aggregation. However, variety of network topologies is limited to too much Pads. Therefore we improve it by adding Pads with proximate operators such as Pooling, Conv2d, DepthwiseConv2d, Pad, etc.

Operators with padding, including Pooling, Conv2d, DepthwiseConv2d, etc. We calculate padding of these operators to keep the shapes of input and output consistent. Take Conv2d with 'SAME' padding for example, given input shape [iN,iH,iW,iC] and other parameters, the output height oH is computed as (2), where pH is padding height, fH is filter height, sH is stride height, and dH is dilation height.

$$oH = (iH + 2 \cdot pH - dH \cdot (fH - 1))/sH \tag{2}$$

Regarding the shapes of layer input and output are consistent, we get:

$$oH = iH \tag{3}$$

Then we associate other parameters, keep it 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.

$$0 \leqslant pH \leqslant fH \tag{4}$$

$$1 \leqslant sH \leqslant Max\_sH \tag{5}$$

$$1 \leqslant dH \leqslant Max\_dH \tag{6}$$

Similarly, weight of parameters 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 (2)-(6), can be generated randomly with less Pads.

IV. Experiment Setup

A. research questions

Six research questions are highlighted as follows.

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

RQ2: How does our MCTS-based search algorithm compare with random search for decision processes?

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

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

RQ5: How does the sub graph defined as block with mutations of block impact the effectiveness of our approach?

RQ6: Whether the exceptions found in our approach are consistent with expectations of operator-level coverage?

B. Setup

We set up our experiments as follows.

Block Corpus. Blocks of corpus in this experiment are depicted as Table III which consist of 22 block in three parts. (1) Unsupported TensorFlow operators in SNPE SDK Reference Guide, including Biasadd and Exp. (2) Sub graphs. Block 21 is inspired by operator fusion and arithmetic optimizer in TensorFlow graph optimization. Block 22 references the concatenation of multiple feature maps in SSD [22]. (3) Operators supported in SNPE SDK Reference Guide. We set

TABLE III
BLOCK CORPUS OF OUR EXPERIMENT

Index	Block Name	Input degree	Inner Edges
1	Add	{2}	N/A
2	Addn	{1,2,3,4,5}	N/A
3	Avgpooling	{1}	N/A
4	Biasadd	{1}	N/A
5	Concat	{1,2,3,4,5}	N/A
6	Conv2d	{1}	N/A
7	DepthwiseConv2d	{1}	N/A
8	Exp	{1}	N/A
9	FusedBatchNorm	{1}	N/A
10	Maxpooling	{1}	N/A
11	Mul	{2}	N/A
12	Pow	{1}	N/A
13	Relu	{1}	N/A
14	Relu6	{1}	N/A
15	Reducemean	{1}	N/A
16	Softmax	{1}	N/A
17	Sub	{2}	N/A
18	Sqrt	{1}	N/A
19	Tanh	{1}	N/A
20	Transpose	{1}	N/A
21	Conv2d+Tanh+Mul +Mul+Mul+Addn	{1}	{(0,1),(0,2),(0,3), (0,4),(1,2),(1,3),(1,4), (2,5),(3,5),(4,5) }
22	Relu+Conv2d+Conv2d +Conv2d+Conv2d+Conv2d +Conv2d+Conv2d+Concat	{1}	{(0,1), (0,2), (0,3), (0,4), (0,5), (0,6), (0,7), (1,8), (2,8), (3,8), (4,8), (5,8), (6,8), (7,8)}

maximum of in-degree or out-degree to 5. The range of out-degree (omitted in Table III) is {0, 1, 2, 3, 4, 5}. TensorFlow API of Blocks in Python is used as defined in SNPE SDK Reference Guide .

Inference Runtime. We conduct experiments on SNPE. CPU runtime(SNPE 1.35, FP32) and DSP runtime (SNPE 1.32, Int8, XiaoMi 9 SnapDragon 855) of SNPE are applied. Test samples contain neural networks models generated in TensorFlow format(.pb file), and their inference results computed by TensorFlow 1.12 (CPU mode). Random inputs, filters and biases are generated from the uniform distribution [-1,1]. Tensor format is NHWC. If SNPEs output of a test sample are inconsistent with TensorFlow's output, the test sample is considered as a failure. Final status of the process contains: Models Conversion Failure (MCF), Inference Failure (IF), Data Comparison Failure (DCF) and Data Comparison Pass (DCP). The threshold of CPU runtime VS TensorFlow comparison is values with relative error less than 1% account for more than 99% in the result tensor. The threshold of DSP runtime VS TensorFlow comparison is values with relative error less than 10% account for more than 70% . The number of exceptions is the sum of exceptions detected both in CPU runtime and DSP runtime. The probability of four models-level mutations is chosen from {0, 0.05, 0.1, 0.3}. The number of neighbors k is chosen from {2, 4, 6, 8}. The probability of rewriting connections p is chosen from {0.5, 0.7, 0.9}. The target mean of types of shapes&parameters is set to 400. For RQ2, RQ3 and RQ4, We make each strategy generate 400 input samples, and each strategy runs 10 times to get 10 sample coverage and inference results.

Some failures are caused by the same defect. To eliminate duplicates, model conversion failures with same error type, error code and failed block are considered as the same fault, data comparison(with TensorFlow) failures with same topology and operators are considered to be duplicated.

V. EVALUATION

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

In order to answer RQ1, we generate approximately 1000 test samples using operators of block corpus in Table III. The number of blocks is chosen uniformly from 1 to 150. We list some typical exceptions as below.

Models Conversion Failure (MCF).

MCF-1 Converter cannot resolve Addn with same inputs. Figure 7(a) shows that, two inputs of addn_outputdata_10006 comes from the same operator, mul_10005 in pb model. SNPE Log: ConverterError: ERROR_TF_ADD_N_NUM_OF_INPUTS: Expected two or more inputs for Addn operation: addn_outputdata_10006, converter cannot resolve at least two inputs. Other operators with multiple-input, such as Add, Concat, Sub, have similar exceptions.

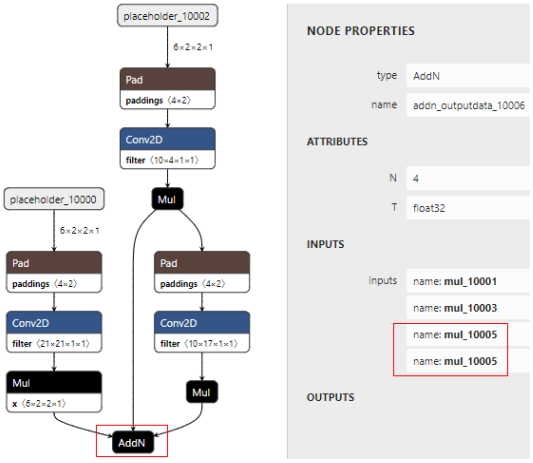
MCF-2 Converter loses output of Conv2d with Biasadd. Addn has three inputs as shown in Figure 7(b) , Conv2d, Biasadd, Relu. The parameter use_bias of Conv2d is False. But the Addn of DLC model shown in Figure 7(c) has 2 inputs. And the converter loses one output of Conv2d. Biasadd is not listed in supported layers of SNPE SDK Reference Guide. But Conv2d with use_bias=0 connect to Biasadd will be convert correctly. So SNPE need to illustrate the limited condition of Biasadd.

Inference Failure(IF).

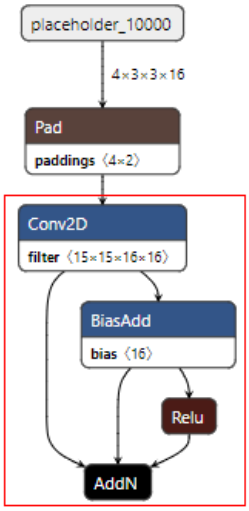
IF-1 DSP only support power is integer. The power of Pow is a floating point: -1.00714. SNPE Log: error_code=902; error_message
=Layer parameter value is invalid in DSP. Layer pow_15200: Only support power is integer. error_component=DSP Runtime; line_no
=631. Power of Pow supports both floats and integers in TensorFlow. And the limitation is not shown by SNPE.

IF-2 Oversized output dimension of Avgpooling whose padding is same. SNPE Log: error_code=902; error_message=Layer parameter value is invalid in DSP. Layer AvgPool2D/AvgPool: Average pooling parameters excluding padding in pool region is not supported for DSP runtime. Parameters: 4x3x3x3 input dims, 1x1x3 output dims, 4294967295x4294967295 padding, 3x3 stride, 1x1 kernel, avg pooling type, not include padding in pool region.; error_component=DSP Runtime; line_no=416. It is suspected that an integer overflow occurs.

IF-3 DSP only supports axis value of 3 in reduce mean. SNPE Log: error_code=900; error_message=Layer is not supported in DSP. Layer reducemean_10000: reduce mean with axes of size 1 only supports axis value of 3 in DSP.



(a) MCF-1: Addn with same inputs in pb model



(b) MCF-2: Addn with three inputs in pb model

Id	Name	Type	Inputs	Outputs	Out Dims	Runtimes	Parameters
0	placeholder_10000:0	data	placeholder_10000:0	placeholder_10000:0	4x3x3x16	A D G C	input_preprocessing: passthrough input_type: default
1	conv2d_nhwc_pad_10000	pad	placeholder_10000:0	conv2d_nhwc_pad_10000:0	4x17x17x16	A D G C	paddings: [[0L, 0L], [7L, 7L], [7L, 7L], [0L, 0L]] mode: constant constant_values: 0
2	conv2d_10000	convolutional	conv2d_nhwc_pad_10000:0	biasadd_10001:0	4x3x3x16	A D G C	padding_x: 0 padding_y: 0 padding mode: zero stride_x: 1 stride_y: 1 num filters: 16 kernel: 15x15 param count: 57k (100%) MACs per inference: 518k (100%)
3	relu_10002	neuron	biasadd_10001:0	relu_10002:0	4x3x3x16	A D G C	a: 0 b: 0 min_clamp: 0 max_clamp: 0 func: relu
4	addn_outputdata_10003	elementwise_op	biasadd_10001:0 relu_10002:0	addn_outputdata_10003:0	4x3x3x16	A D G C	operation: sum

(c) MCF-2: Network architecture of DLC model extracted from SNPE log

Fig. 7. MCF-1: Addn with same inputs in pb model, and MCF-2: Converter loses output of Conv2d with Biasadd

reduce mean with axis of size 1 only supports each axis in TensorFlow. And the limitation is not shown by SNPE.

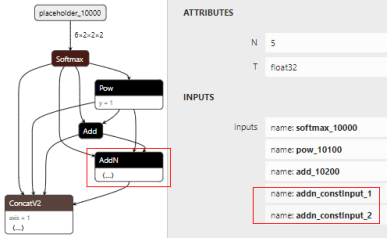
Data Comparison Failure(DCF)

DCF-1 Data Comparison Failure of Avgpooling. Avgpooling with SAME padding, and its other parameters are shown in Figure 8. The inference result of CPU runtime VS TensorFlow comparison is values with relative error less than 1% account for 93.89%, and does not reach the threshold 99%.

Id	Name	Type	Inputs	Outputs	Out Dims	Parameters
0	input_1:0	data	input_1:0	input_1:0	4x3x3x3	input_preprocessing: passthrough input_type: default
1	AvgPool2D/AvgPool	pooling	input_1:0	AvgPool2D/AvgPool:0	4x1x1x3	pool size x: 1 pool size y: 1 stride x: 3 stride y: 3 padding x: -1 padding y: -1 pool_type: POOL_AVG MACs per inference: 3 (100%)
2	pad	pad	AvgPool2D/AvgPool:0	pad:0	4x3x5x3	paddings: [[0L, 0L], [1L, 1L], [2L, 2L], [0L, 0L]] mode: constant constant_values: 0

Fig. 8. DCF-1: Avgpooling of DLC model extracted from SNPE log

DCF-2 Data Comparison Failure of Addn. Addn shown in Figure 9 has five inputs, in which two of them are constant inputs. And the Addn of DLC model shown in Figure9(a) has five inputs. And In Figure 9(b), the converter loses the two constant inputs for Addn. Addn is one of five inputs of Concat. The result of Concat in CPU runtime VS TensorFlow comparison is values with relative error less than 1% account for 80%, and does not reach the threshold 99%. The difference between the 20% numbers is 2, which is the sum of two constant inputs of Addn.



(a) Addn with five inputs in pb model

Id	Name	Type	Inputs	Outputs	Out Dims	Runtimes	Parameters
0	placeholder_10000:0	data	placeholder_10000:0	placeholder_10000:0	6x2x2x2	A D G C	input_preprocessing: passthrough input_type: default
1	softmax_10000	softmax	placeholder_10000:0	softmax_10000:0	6x2x2x2	A D G C	
2	pow_10100	power	softmax_10000:0	pow_10100:0	6x2x2x2	A D G C	scale: 1.0 shift: 0.0 power: 1.0
3	add_10200	elementwise_op	softmax_10000:0 pow_10100:0	add_10200:0	6x2x2x2	A D G C	operation: sum
4	addn_10300	elementwise_op	softmax_10000:0 pow_10100:0 add_10200:0	addn_10300:0	6x2x2x2	A D G C	operation: sum
5	concat_constInput0_10400:0	constant		concat_constInput0_10400:0	6x2x2x2	A D G C	
6	concat_outputdata_10400	concatenation	softmax_10000:0 pow_10100:0 add_10200:0 addn_10300:0 concat_constInput0_10400:0	concat_outputdata_10400:0	6x10x2x2	A D G C	axis: 1

(b) Addn of DLC model extracted from SNPE log

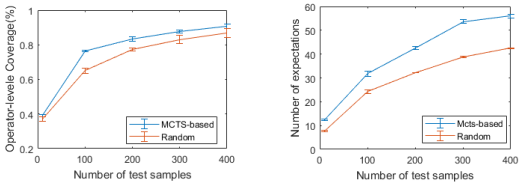
Fig. 9. DCF-2: Data Comparison Failure of Addn

Answer to RQ1: Guided by operator-level coverage, our approach is effective to detect various exceptions in models conversion and inference process for the DL inference engine.

B. RQ2: How does our MCTS-based search algorithm compare 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 search for block chooser. The block number of each neural network is generated from the uniform distribution [5, 20]. In MCTS-based block chooser, $tc1$ is set to 10, $tc2$ is set to 1 and e is set to $1/\sqrt{2}$.

Operator-level coverage. The operator-level coverage of two search algorithms are 90.6% and 87.1% respectively, and MCTS-based search, on average, covers 3.5% more operator-level coverage than random search where number of test sample is 400 as demonstrated in Figure 10(a).



(a) Operator-level coverage of inputs (b) Comparison of exceptions

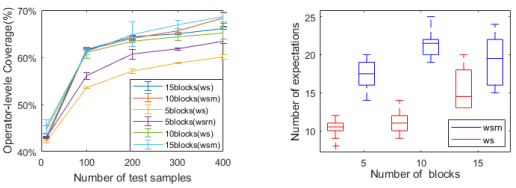
Fig. 10. Operator-level coverage of inputs and Comparison of exceptions with MCTS-based search and random search of block chooser

Inference results. We measure the exceptions of inference results for each search algorithm. Figure 10(b) shows the trend of number of expectations. MCTS-based search, on average, find 56.2 expectations and random search find 42.4 when the number of test samples reach 400.

Answer to RQ2: The MCTS-based block chooser outperforms random-based block chooser in boosting operator-level coverage(3.5% more) and detecting exceptions (13.8 more).

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

In order to answer RQ3, we evaluate operator-level coverage and inference results using two stochastic network generation strategies: (1) WS and RN model together , each model generates half test samples (2) WS model only. Five input shapes can be chosen uniformly for each neural networks. To avoid disturbing the coverage and inference result of random graph models, mutations are cancelled. And the block number of each neural network is set to 5, 10 and 15 respectively.



(a) Operator-level coverage of inputs (b) Comparison of exceptions

Fig. 11. Comparison of Operator-level coverage and exceptions detected with inputs generated by WS and WS+RN

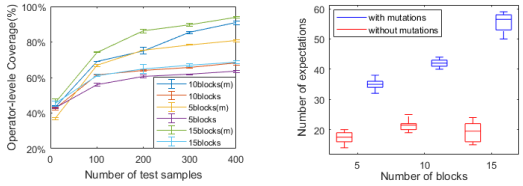
Operator-level coverage. As shown in Figure 11(a), the operator-level coverage of two search algorithms are 60.01% and 63.38% for 5 blocks, 66.13% and 68.28% for 10 blocks, 65.23% and 68.6% for 15 blocks respectively. We can make two key observations from the results. First, WS and RN model together, on average, covers 2.9% more operator-level coverage than WS model as demonstrated. Second, as the block number increases, the two strategies cover more. This is intuitive as a higher value of blocks makes it increasingly harder to cover more topologies and types of shapes&parameters.

Inference results. We measure the exceptions of inference results for each strategy. Figure 11(b) shows the detailed box-plot results. The mean number of expectations of are 10.3 and 17.5 for 5 blocks, 11.2 and 21.4 for 10 blocks, 15.4 and 19.5 for 15 blocks respectively. In general, WS+RN model is more efficient of finding exceptions as well as increasing blocks of stochastic networks.

Answer to RQ3: Through applying the RN model to stochastic network generation strategy, on average, 7.2 more exceptions for each strategy can be found as well as operator-level coverage increasing.

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

In order to answer RQ4, we evaluate operator-level coverage and inference results of generation with mutations and generation without mutations. In generation without mutations, 5 input shapes can be chosen uniformly for each neural networks without mutations. The block number of neural network is set as 5, 10 and 15.



(a) Operator-level coverage of inputs (b) Comparison of exceptions

Fig. 12. Comparison of Operator-level coverage and exceptions detected with inputs generated with mutations and generated without mutations

Operator-level coverage. The operator-level coverage of two search algorithms are 63.38% and 80.71% for 5 blocks, 68.28% and 90.81% for 10 blocks, 68.6% and 93.87% for 15 blocks respectively. Generation with mutations, on average, covers 21.6% more operator-level coverage than generation without mutations as demonstrated in Figure 12(a).

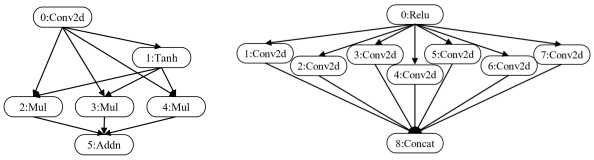
Inference results. We measure the exceptions of inference results for each strategy. Figure 12(b) shows the detailed box-plot results. The mean number of expectations are 17.5 and 35 for 5 blocks, 21.4 and 42.1 for 10 blocks, 19.5 and 55.7 for 15 blocks respectively. In general, we can see that generation with mutations is more efficient of finding exceptions(on average, 24.8 more exceptions) as well as increasing blocks of stochastic networks.

Answer to RQ4: Our mutants are useful to generate new valid test inputs with original graph or sub graph, by up to a 21.6% more operator-level coverage and 24.8 more exceptions captured on average.

E. RQ5: How does the sub graph defined as block with mutations of block impact the effectiveness of our approach?

In order to answer RQ5, we generate 100 mutated sub graphs (MS) to evaluate effectiveness of sub graph in block corpus (shown in Figure 13). However, the suitable and targeted sub graph should be derived from those optimizations designed in SNPE, but cannot be got from SNPE SDK documents currently. The number of blocks of a network is

chosen from $\{2, 3\}$. We analyze mutated sub graph inference results respectively as below.

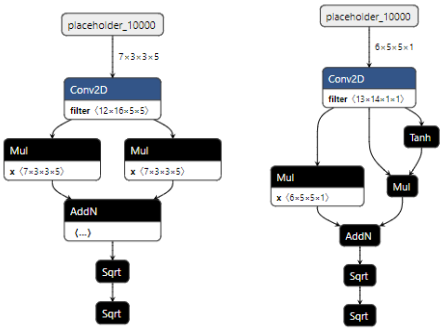


(a) Sub graph of block 21 in block corpus (b) Sub graph of block 22 in block corpus

Fig. 13. architecture of sub graphs defined in block corpus

MS-1. The success rate of block inference result is 20%. The major error types are Inference Failure (error_code=910 only) and Data Comparison Failure. Figure 14 shows two typical mutated sub graph.

MS-2. The success rate of block inference result is 52%. The major error types are also Inference Failure (DSP only, error_code=910) and Data Comparison Failure (both DSP and CPU runtime). Figure 15 shows two typical mutated sub graph and their inference result that run failed in CPU runtime or DSP runtime.



(a) MDelete Tanh and Mul, Data Comparison Failure of Concat on CPU (7.62%) and Inference Failure on DSP (error_code=910)

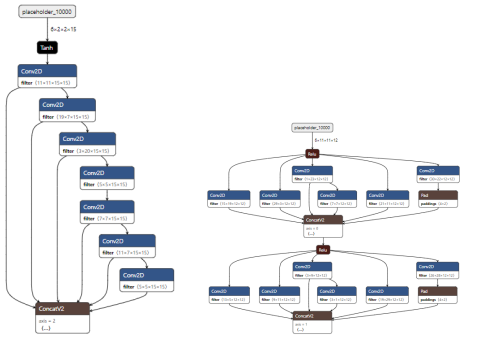
(b) Delete Addn and Mul, Data Comparison Failure of Concat on CPU (12.67%) and Inference Failure on DSP (error_code=910)

Fig. 14. MS-1 Inference result of mutated sub graph of block 21 in block corpus

Answer to RQ5: To some extent, some specific sub graph and their mutants are hard to generate by matching operator to nodes of random graphs. Sub graph defined in block corpus could increase the probability of occurrence of specific sub graph. And sub graph and their mutated sub graph can cover those specific topologies designed for optimizations in DL inference engine accurately. sub graph defined for blocks is needed to enhance diversity of NNs.

F. RQ6: Whether the exceptions found in our approach are consistent with expectations of operator-level coverage?

In order to answer RQ6, we analyze relation between the typical exceptions and operator-level coverage.



(a) Delete Relu, Data Comparison Failure of Concat on DSP (35.08%)

(b) MS-2 Delete a Conv2d and reconnect an edge between Relu and Conv2d. Data comparison failure of Concat on DSP (58.80%).

Fig. 15. Inference result of mutated sub graph of block 22 in block corpus

Type of Operator Coverage. Take MCF-2 for example, Biasadd is not listed in supported layers of SNPE SDK Reference Guide. But Conv2d with use_bias=0 connect to Biasadd will be convert correctly. And limitations of topologies should be described.

Single Edge Coverage is usually associated with **Input Degree Coverage** or (Output Degree Coverage), such as multi-input operators (i.e. Addn of DCF-2) and multi-output operators (i.e. Conv2d of MCF-2).

Multiple Edges Coverage is usually associated with Input degree of Operator Coverage, such as MCF-1. In-degree of multi-input operators (i.e. Add, Addn, Concat, etc) in SNPE are different with TensorFlow.

Shapes&Parameters Coverage. We note that some parameters range of operators supported in SNPE are not consistent with TensorFlow, such as power of Pow in IF-1, Reducemean in IF-3 and Conv2d in IF-4. And the limitations are not declares in the SNPE SDK Reference Guide. Some results of operators with specific value of parameters or tensor shapes are unexpected in comparison with TensorFlow, such as Avg-pooling in DCF-1.

Answer to RQ6: In summary, exceptions we found 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, these metrics may trigger various types of exceptions in SNPE.

VI. Related Work

Fuzz and Mutation Testing. Fuzzing is a widely used technique for exposing defects in DL system. Guo et al. [23] proposed the first differential fuzzing framework for DL systems. TensorFuzz proposed by Odena et al. [5], 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 neural networks and their quantized versions. Pei et al. presented DeepXplore [6] which proposed a white-box differential testing technique to generate test inputs for DL system. Wicker et al. [24] proposed

feature-guided test generation. They transformed the problem of finding adversarial examples into a two-player turn-based stochastic game. Ma et al. [7] 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. [25] proposed five mutation operators for DNNs and evaluated properties of mutation . Xie et al. [26] presented a metamorphic transformation based coverage guided fuzzing technique, DeepHunter, which leverages both neuron coverage and coverage criteria presented by DeepGauge [27].

Test Coverage. Unlike traditional software, code coverage is seldom a demanding criterion for ML testing, since the decision logic of an ML model is not written manually but rather it is learned from training data [28]. In the study of Pei et al. [6], 100 % traditional code coverage is easily achieved by a single randomly chosen input. Pei et al. [6] 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. [29] proposed layer-level Coverage, which considers the top hyperactive neurons and their combinations to characterise the behaviours of a DNN. Du et al. [25] first proposed State-level Coverage to capture the dynamic state transition behaviours of deep neural network. Li et al. [30] pointed out the limitations of structural coverage criteria for deep networks caused by the fundamental differences between neural networks and human-written programs. DeepCover [31] proposes the test cirteria [32] for DNNs, adapted from the MC/DC test criteria of traditional software.

VII. Conclusion

In this paper, we firstly proposed a novel graph-based fuzzer for DL inference engine fuzz testing. We propose graph-based stochastic neural network generator, and a set of mutations for NNs generation. We also introduce a coverage criterion, operator-level coverage based on graph theory, and a search algorithm using MCTS for decision processes of block chooser. Our evaluation shows that our method could generates various NNs and more efficiently finds more incorrect corner case behaviors.

More details of our method and experiments can be found on https://github.com/gbftdlie/Graph-based-fuzz-testing.

References

[1] "Snpe," 2018. [Online]. Available: https://developer.qualcomm.com/docs/snpe/overview.html

[2] "Hiai," 2018. [Online]. Available: https://developer.huawei.com/consumer/en/doc/2020315

[3] "Tensorflow lite," 2017. [Online]. Available: https://tensorflow.google.cn/lite

[4] "Mnn," 2020. [Online]. Available: https://github.com/alibaba/MNN

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

[6] 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.

[7] 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 *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[8] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1284–1293.

[9] "Supported network layers," 2019. [Online]. Available: https://developer.qualcomm.com/docs/snpe/network_layers.html

[10] "Caffe," 2019. [Online]. Available: https://caffe.berkeleyvision.org/

[11] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[12] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[13] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[14] 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

[15] "Tensorflow," 2020. [Online]. Available: https://github.com/TensorFlow/TensorFlow

[16] "Tensorflow graph optimization with grappler," 2019. [Online]. Available: https://www.TensorFlow.org/guide/graph_optimization

[17] T. S. Rasmus Munk Larsen, "Tensorflow graph optimizations," 2019. [Online]. Available: https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

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

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

[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-worldnetworks," *nature*, vol. 393, no. 6684, p. 440, 1998.

[22] 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.

[23] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[24] 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.

[25] W. Shen, J. Wan, and Z. Chen, "Munn: Mutation analysis of neural networks," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 108–115.

[26] 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.

[27] 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.

[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] 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.

[30] 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 (ICSE-NIER)*. IEEE, 2019, pp. 89–92.

[31] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.

[32] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.