Mutation-based Fault Localization
of Deep Neural Networks

Ali Ghanbari
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
alig@iastate.edu

Deepak-George Thomas
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
dgthomas@iastate.edu

Muhammad Arbab Arshad
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
arbab@iastate.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, Iowa, USA
hridesh@iastate.edu

Abstract—**Deep neural networks (DNNs) are susceptible to bugs, just like other types of software systems. A significant uptick in using DNN, and its applications in wide-ranging areas, including safety-critical systems, warrant extensive research on software engineering tools for improving the reliability of DNN-based systems. One such tool that has gained significant attention in the recent years is *DNN fault localization*. This paper revisits *mutation-based fault localization* in the context of DNN models and proposes a novel technique, named deepmufl, applicable to a wide range of DNN models. We have implemented deepmufl and have evaluated its effectiveness using 109 bugs obtained from StackOverflow. Our results show that deepmufl detects 53/109 of the bugs by ranking the buggy layer in top-1 position, outperforming state-of-the-art static and dynamic DNN fault localization systems that are also designed to target the class of bugs supported by deepmufl. Moreover, we observed that we can halve the fault localization time for a pre-trained model using *mutation selection*, yet losing only 7.55% of the bugs localized in top-1 position.**

Index Terms—**Deep Neural Network, Mutation, Fault Localization**

I. INTRODUCTION

Software bugs [1] are a common and costly problem in modern software systems, costing the global economy billions of dollars annually [2]. Recently, data-driven solutions have gained significant attention for their ability to efficiently and cost-effectively solve complex problems. With the advent of powerful computing hardware and an abundance of data, the use of deep learning [3], which is based on deep neural networks (DNNs), has become practical. Despite their increasing popularity and success stories, DNN models, like any other software, may contain bugs [4], [5], [6], [7], which can undermine their safety and reliability in various applications. Detecting DNN bugs is *not* easier than detecting bugs in traditional programs, *i.e.*, programs without any data-driven component in them, as DNNs depend on the properties of the training data and numerous hyperparameters [8]. Mitigating DNN bugs has been the subject of fervent research in recent years, and various techniques have been proposed for testing [9], [10], fault localization [11], [12], and repair [13], [14] of DNN models.

Fault localization in the context of traditional programs has been extensively studied [15], with one well-known approach being *mutation-based fault localization* (MBFL) [16], [17].

This approach is based on mutation analysis [18], which is mainly used to assess the quality of a test suite by measuring the ratio of artificially introduced bugs that it can detect. MBFL improves upon the more traditional, lightweight *spectrum-based fault localization* [19], [20], [21], [22], [23], [24] by uniquely capturing the relationship between individual statements in the program and the observed failures. While both spectrum-based fault localization [25], [26] and mutation analysis [27], [28], [29] have been studied in the context of DNNs, to the best of our knowledge, MBFL for DNNs has not been explored by the research community, yet the existing MBFL approaches are not directly applicable to DNN models.

This paper revisits the idea of MBFL in the context of DNNs. Specifically, we design, implement, and evaluate a technique, named deepmufl, to conduct MBFL in pre-trained DNN models. The basic idea behind deepmufl is derived from its traditional MBFL counterparts, namely, Metallaxis [30] and MUSE [17], that are based on measuring the impact of mutations on passing and failing test cases (see §II for more details). In summary, given a pre-trained model and a set of data points, deepmufl separates the data points into two sets of "passing" and "failing" data points (test cases), depending on whether the output of the model matches the ground-truth. deepmufl then localizes the bug in two phases, namely *mutation generation phase* and *mutation testing/execution phase*. In mutation generation phase, it uses 79 mutators, a.k.a. mutation operators, to systematically mutate the model, *e.g.*, by replacing activation function of a layer, so as to generate a pool of mutants, *i.e.*, model variants with seeded bugs. In mutation testing phase, deepmufl feeds each of the mutants with passing and failing data points and compares the output to the output of the original model to record the number of passing and failing test cases that are impacted by the injected bugs. In this paper, we study two types of impacts: *type 1* impact, *à la* MUSE, which tracks only fail to pass and pass to fail, and *type 2* impact, like Metallaxis, which tracks changes in the actual output values. deepmufl uses these numbers to calculate *suspiciousness values* for each layer according to MUSE, as well as two variants of Metallaxis formulas. The layers are then sorted in descending order of their suspiciousness values for the developer to inspect.

We have implemented deepmufl on top of Keras [31], and it

supports three types of DNN models for regression, as well as classification tasks that must be written using `Sequential` API of Keras: fully-connected DNN, convolutional neural network (CNN), and recurrent neural network (RNN). Extending deepmufl to other libraries, *e.g.*, TensorFlow [32] and PyTorch [33], as well as potentially to other model architectures, *e.g.*, functional model architecture in Keras, is a matter of investing engineering effort on the development of new mutators tailored to such libraries and models. Since the current implementation of deepmufl operates on pre-trained models, its scope is limited to *model bugs* [7], *i.e.*, bugs related to activation function, layer properties, model properties, and bugs due to missing/redundant/wrong layers (see §VI).

We have evaluated deepmufl using a diverse set of 109 Keras bugs obtained from StackOverflow. These bugs are representatives of the above-mentioned model bugs, in that our dataset contains examples of each bug sub-category at different layers of the models suited for different tasks. For example, concerning the sub-category *wrong activation function* model bug, we have bugs in regression and classification fully-connected DNN, CNN, and RNN models that have wrong activation function of different types (*e.g.*, ReLU, softmax, *etc.*) at different layers. For 53 of the bugs, deepmufl, using its MUSE configuration, pinpoints the buggy layer by ranking it in top-1 position. We have compared deepmufl's effectiveness to that of state-of-the-art static and dynamic DNN fault localization systems Neuralint [12], DeepLocalize [11], DeepDiagnosis [8], and UMLAUT [34] that are also designed to detect model bugs. Our results show that, in our bug dataset, deepmufl, in its MUSE configuration, is 77% more effective than DeepDiagnosis, which detects 30 of the bugs.

Despite this advantage of deepmufl in terms of effectiveness, since it operates on a pre-trained model, it is slower than state-of-the-art DNN fault localization tools from an end-user's perspective. However, this is mitigated, to some extent, by the fact that similar to traditional programs, one can perform *mutation selection* [35] to curtail the mutation testing time: we observed that by randomly selecting 50% of the mutants for testing, we can still find 49 of the bugs in top-1 position, yet we halve the fault localization time after training the model.

In summary, this paper makes the following contributions.

- **Technique**: We develop MBFL for DNN and implement it in a novel tool, named deepmufl, that can be uniformly applied to a wide range of DNN model types.
- **Study**: We compare deepmufl to state-of-the-art static and dynamic fault localization approaches and observed:
  - In four configurations, deepmufl outperforms other approaches in terms of the number of bugs that appear in top-1 position and it detects 21 bugs that none of the studied techniques were able to detect.
  - We can halve the fault localization time for a pre-trained model by random mutation selection without significant loss of effectiveness.
- **Bug Dataset**: We have created the largest curated dataset of *model bugs*, comprising 109 Keras models rang-

ing from regression to classification and fully-connected DNN to CNN and RNN.

Paper organization. In the next section, we review concepts of DNNs, mutation analysis, and MBFL. In §III, we present a motivating example and discusses how deepmufl works under the hood. In §IV, we present technical details of the proposed approach, before discussing the scope of deepmufl in §V. In §VI, we present the results of our experiments with deepmufl and state-of-the-art DNN fault localization tools from different aspects. We discuss threats to validity in §VII and conclude the paper in §IX.

Data availability. The source code of deepmufl and the data associated with our experiments are publicly available [36].

II. BACKGROUND

A. Mutation Analysis

Mutation analysis [18], is a program analysis method for assessing the quality of the test suite. It involves generating a pool of program variants, *i.e.*, *mutants*, by systematically mutating program elements, *e.g.*, replacing an arithmetic operator with another, and running the test suite against the mutants to check if the output of the mutated program is different from that of the original one; if different, the mutant is marked as *killed*, otherwise as *survived*. A mutant might survive because it is semantically equivalent to the original program, hence the name *equivalent* mutant. Test suite quality is assessed by computing a *mutation score* for the test suite, which is the ratio of killed mutants over the non-equivalent survived mutants. Mutation score indicates how good a test suite is in detecting real bugs [37]. In addition to its original use, mutation analysis has been used for many other purposes [38], such as fault localization [16], [17], automated program repair [39], [40], test generation [41], [42] and prioritization [43], program verification [44], [45], *etc*.

B. Mutation-based Fault Localization

Mutation-based fault localization (MBFL) uses mutation analysis to find bugs. In this section, we review two major approaches to MBFL, namely Metallaxis [30] and MUSE [17]. Both of these approaches are implemented in deepmufl. The reader is referred to the original papers [30], [17] for examples explicating the rationale behind each approach.

1) Metallaxis: Metallaxis [30] posits that mutants generated by mutating the same program element are likely to exhibit similar behaviors and mutants generated by mutating different program elements are likely to exhibit different behaviors. Since a fault itself can also be viewed as a mutant, it is expected to behave similar to other mutants generated by mutating that same buggy program element and can be located by examining the mutants based on this heuristic. Metallaxis assumes that the mutants impacting the test outputs, or their error messages, *e.g.*, stack trace, as *impacting the tests*. Thus, mutants impacting failing test cases might indicate that their corresponding code elements is the root cause of the test failures, while mutants impacting passing test cases might indicate that their corresponding code elements are correct.

Once the number of impacted passing and failing test cases are calculated, Metallaxis uses a fault localization formula to calculate suspiciousness values for each element.

Metallaxis fault localization formula can be viewed as an extension to that of spectrum-based fault localization, by treating all mutants impacting the tests as covered elements while the others as uncovered elements. Specifically, the maximum suspiciousness value of the mutants of a corresponding code element is returned as the suspiciousness value of the code element. More concretely, assuming we are using SBI formula [46], suspiciousness value for a program element e, denoted $s(e)$, is calculated as follows.

$$s(e) = \max_{m \in M(e)} \left(\frac{|T_f(m,e)|}{|T_f(m,e)| + |T_p(m,e)|} \right), \qquad (1)$$

where $M(e)$ denotes the set of all mutants targeting program element e, $T_f(m,e)$ is the set of failing test cases that are impacted by the mutant m, while $T_p(m,e)$ denotes the set of passing test cases that are impacted by m. In this definition, and in the rest of the paper, the notation $|\cdot|$ represents the size of a set. Alternatively, had we used Ochiai [47], Metallaxis suspiciousness formula would be modified as follows.

$$s(e) = \max_{m \in M(e)} \left(\frac{|T_f(m,e)|}{\sqrt{(|T_f(m,e)| + |T_p(m,e)|)|T_f|}} \right), \quad (2)$$

where T_f denotes the set of all failing tests cases.

2) MUSE: MUSE [17] is based on the assumption that mutating a faulty program element is likely to impact more failed test cases than passing test cases by "fixing" it, while mutating a correct program element is likely to impact more passing test cases than failing test cases by breaking it. The notion of "impacting test cases" in MUSE, unlike Metallaxis, is more rigid, in that it refers to making passing test cases fail, *vice versa*. Once the number of impacted failing and passing test cases are identified, *suspiciousness values* can be calculated using the following formula.

$$s(e) = \frac{1}{|M(e)|} \Sigma_{m \in M(e)} \left(\frac{|T_f(m,e)|}{|T_f|} - \alpha \frac{|T_p(m,e)|}{|T_p|} \right), \quad (3)$$

where T_p denotes the set of all passing tests cases and α is a constant used to balance the two ratios that is defined to be $\frac{|F \leadsto P|}{|T_f|} \times \frac{|T_p|}{|P \leadsto F|}$. In the latter definition, $F \leadsto P$ denotes the set of failing test cases that pass due to some mutation, while $P \leadsto F$ denotes the set of passing test cases that fail as a result of some mutation.

C. Deep Neural Networks

A neural network is intended to compute a function of the form $\mathbb{R}^m \longrightarrow \mathbb{R}^n$, where m, n are positive integers. A neural network is often represented as a weighted directed acyclic graph arranged in layers of three types, *i.e.*, *input layer*, one or more *hidden layers*, and an *output layer*. Input and output layers output linear combination of their inputs, while hidden layers can be viewed as more complex computational units, *e.g.*, a *non-linear unit*, *convolutional unit*, or a *batch normalization unit*. A non-linear unit is composed of

neurons, functions applying a non-linear *activation function*, *e.g.*, rectified linear unit (ReLU), tanh, or sigmoid, on the weighted sum of their inputs. A convolutional layer, calculates the convolution between the vector of the values obtained from the previous layer and a learned kernel matrix. Lastly, a batch normalization layer, normalizes the vector of values obtained from the previous layer *via* centering or re-scaling. A neural network with two or more hidden layers is referred to as a *deep neural network* (DNN).

III. MOTIVATING EXAMPLE

In this section, we first describe how deepmufl helps programmers detect and fix bugs by presenting a hypothetical use case scenario and then motivate the idea behind deepmufl by describing the details of how deepmufl works, under the hood, on the example developed in the use case story.

Courtney is a recent college graduate working as a junior software engineer at an oil company, which frequently makes triangular structures, made of epoxy resin, of varying sizes to be used under the water. The company needs to predict with at least 60% confidence that a mold of a specific size will result in an epoxy triangle after it has been dried, and potentially shrunk, and it does not need to spend time on cutting and/or sanding the edges. Over time, through trial and error, the company has collected 1,000 data points of triangle edge lengths and whether or not a mold of that size resulted in a perfect triangle. Courtney's first task is to write a program that given three positive real numbers a, b, and c, representing the edge lengths of the triangle mold, determines if the mold will result in epoxy edges that form a perfect triangle. As a first attempt, she writes the program shown in Listing 1.

```
1  # load 994 of the data points as X_train and y_train
2  # ...
3  model = Sequential()
4  model.add(Dense(2, activation='relu'))
5  model.add(Dense(2, activation='relu'))
6
7  model.compile(loss='sparse_categorical_crossentropy',
8      optimizer='adam', metrics=['accuracy'])
9  model.fit(X_train, y_train, epochs=100, validation_split=0.1)
```

Listing 1: Courtney's first attempt

The program uses 994 out of 1,000 data points for training a model. After testing the model on the remaining 6 data points, she realizes that the model achieves no more than 33% accuracy. Fortunately, Courtney uses an IDE equipped with a modern DNN fault localization tool, named deepmufl, which is known to be effective at localizing bugs that manifest as stuck accuracy/loss. She uses deepmufl, with its default settings, *i.e.*, Metallaxis with SBI, to find the faulty part of her program. The tool receives the fitted model in .h5 format [48] together with a set of testing data points T and returns a ranked list of model elements; layers, in this case. After Courtney provides deepmufl with the model saved in .h5 format and the 6 testing data points that she had, within a few seconds, the tool returns a list with two items, namely Layer 2 and Layer 1, corresponding to the lines 5 and 4, respectively, in Listing 1. Once she navigates

Table 1: An example of how deepmufl uses Metallaxis' default formula to localize the bug in the model of Listing 1

Layer	Neuron	Mutant	Impact? T1	T2	T3	T4	T5	T6	$\frac{|T_f(m,e)|}{|T_f(m,e)|+|T_p(m,e)|}$	max
L1	N1	M1: weights / 2					•		0	0
		M2: bias / 2					•		0	
		M3: relu → softmax					•	•	0	
	N2	M4: weights / 2					•		0	
		M5: bias / 2					•		0	
		M6: relu → softmax					•	•	0	
L2	N3	M7: weights / 2			•	•		•	0.67	1
		M8: bias / 2		•	•		•		0.67	
		M9: relu → softmax	•	•	•				1	
	N4	M10: weights / 2			•	•		•	0.67	
		M11: bias / 2		•			•		0.5	
		M12: relu → softmax	•	•	•				1	

to the details about `Layer 2`, she receives a ranked list with 5 elements, *i.e.*, `Mutant 12: replaced activation function 'relu' with 'softmax'`, ..., `Mutant 10: divided weights by 2`, `Mutant 11: divided bias by 2`. By seeing the description of `Mutant 12`, Courtney immediately recalls her machine learning class wherein they were advised that in classification tasks they should use *softmax* as the activation function of the last layer. She then changes the activation function of the last layer at Line 5 of Listing 1 from `relu` to `softmax`. By doing so, the model achieves an accuracy of 67% on the test dataset, and similarly on a cross-validation, exceeding the expectations of the company.

We now describe how deepmufl worked, under the hood, to detect the bug *via* Metallaxis' default formula. Figure 1 depicts the structure of the model constructed and fitted in Listing 1. Each edge is annotated with its corresponding weight and the nodes are annotated with their bias values. The nodes are using ReLU as the activation function. In this model, the output T is intended to be greater than the other output if a, b, and c form a triangle, and $\sim T$ should be greater than or equal to the other output, otherwise.
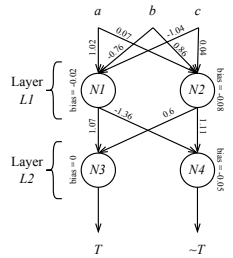

Fig. 1: Model structure built and fitted in Listing 1

Table 1 shows an example of how deepmufl localizes the bug in the model depicted in Figure 1. In the first two columns, the table lists the two layers, and within each layer, the neurons. For each neuron three mutators are applied, *i.e.*, halving weight values, halving bias value, and replacing the activation function. More mutators are implemented in deepmufl, but here, for the sake of simplicity, we only focus on 3 of them and also restrict ourselves to only one activation function replacement, *i.e.*, ReLU vs. softmax.

As we saw in Courtney's example, she had a test dataset T with 6 data points which initially resulted in 33% accuracy. These six data points are denoted T1, ..., T6 in Table 1, where correctly classified ones are colored green, whereas misclassified data points are colored rose. deepmufl generates 12 mutants for the model of Figure 1, namely, M1, ..., M12. Each mutant is a variant of the original model. For example, M1 is the same as the model depicted in Figure 1, except the weights of the
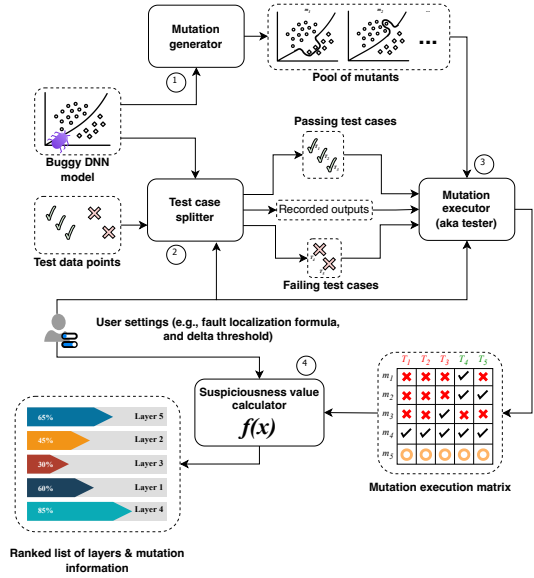

Fig. 2: deepmufl architecture. Processes are denoted using solid round rectangles, while data and artifacts are represented as dotted round rectangles. Arrows represent flow of control and information.

incoming edges to neuron N1 are halved, *i.e.*, 0.51, -0.38, and -0.52 from left to right, while M9 is the same as the model depicted in Figure 1, except that the activation functions for N3 and N4 are `softmax` instead of `relu`. After generating the mutants, deepmufl applies each mutant on the inputs T1, ..., T6 and compares the results to that of the original model. For each data point T1, ..., T6, if the result obtained from each mutant M1, ..., M12 is different from that of the original model, we put a bullet point in the corresponding cell. For example, two bullets points in the row for M3 indicate that the mutant misclassifies the two data points that used to be correctly classified, while other data points, *i.e.*, T1, ..., T4, are misclassified as before. Next, deepmufl uses SBI formula [46] to calculate suspiciousness values for each mutant $m \in \{M1, ..., M12\}$, individually. These values are reported under the one but last column in Table 1. Lastly, deepmufl takes the maximum of the suspiciousness values of the mutants corresponding to a layer and takes it as the suspiciousness value of that layer (c.f. Eq. 1 in §II). In this particular example, layer L1 gets a suspiciousness value of 0, while L2 gets a suspiciousness value of 1. Thus, deepmufl ranks L2 before L1 for user inspection and for each layer it sorts the mutants in descending order of their suspiciousness values, so that the user will understand what change impacted most the originally correctly classified data points. In this case, M12 and M9 wind up at the top of the list, and as we saw in Courtney's story, the information associated with the mutations helped fixing the bug.

IV. PROPOSED APPROACH

Our technique deepmufl comprises four components: (1) mutation generator, (2) test case splitter, (3) mutation executor/tester, and (4) suspiciousness calculator. Figure 2 depicts these components as processes, numbered accordingly, taking

inputs and producing outputs. Mutation generator (marked ①) in Figure 2), applies 79 mutators on all the layers of the input buggy DNN, so as to generate a pool of mutants, *i.e.*, variants of the original, buggy DNN model with small perturbations, *e.g.*, replacing activation function of a layer. Test case splitter (marked ② in the figure), applies the original buggy DNN on a given set of test data points, *i.e.*, test cases (or input values) paired with their expected output values, so as to partition the set into two subset, namely *passing test cases* and *failing test cases*. Passing test cases are referred to as those input values for which the expected output matches that of produced by the original model, whereas failing test cases are referred to as those input values for which the expected output does not match that of produced by the model. This component also stores the output of the original model on each of the test cases. Next, mutation executor (which is also called mutation tester, marked ③ in the figure) applies the generated mutants on each of the passing and failing test cases and the results are compared to that of the original model recorded in the previous step. This amounts to a mutation execution matrix that is used to calculate suspiciousness values for each layer in the model (marked ④ in the figure). The user may instruct deepmufl to use a specific fault localization formula, *e.g.*, MUSE or Metallaxis with SBI or Ochiai, for calculating suspiciousness values. The layers are then ranked based on the calculated suspiciousness values for user inspection. The ranked list is accompanied with the information about the mutations conducted on each layer to facilitate debugging.

A. Mutation Generator

Mutation generator component receives the original, buggy DNN model and generates as many variants of the model, *i.e.*, mutants, as possible, by systematically mutating every elements of the input model. This component implements 79 mutators. Mutators can be viewed as transformation operators that when applied to a given element, *e.g.*, a neuron or a layer, in the model, returns a new variants of the model with that particular element mutated. Table 2 lists all the mutators implemented in deepmufl, the types of model elements on which they can operate, and the way each mutator affects the target element. These mutators are inspired by the ones implemented in the existing mutation analysis systems, *e.g.*, [27], [28], [29], [49], [50], [51], [52], [53], to name a few. Ma *et al.* [29], and Hu *et al.* [28], define so-called *model-level* mutators that also operate on pre-trained models. Direct reuse of all of their mutators was not possible, as those mutators depend on random values which would introduce a source of non-determinism in deepmufl: mutating the same model element with random values, *e.g.*, Gaussian fuzzing, as in [29], would yield a different mutant each time, making deepmufl to produce different outputs on each run for the same model. In general, as far as MBFL is concerned, using variable values (whether it is deteministic or not), instead of the current hard-coded ones, for mutation of weights would not bring about any benefit, as the goal here is to *break* the model in some way and observe its impact on originally failing and passing test cases.

We argue that not all model bugs could be emulated using mutators at the level of pre-trained models, *e.g.*, missing batch normalization, but the mutators listed in Table 2 are sufficient for emulating a subset of such bugs, *e.g.*, wrong activation function or missing/redundant layer. Please see §V for a more detailed discussion on supported bugs.

Mutation generation in deepmufl is done directly on the trained model and there is no need for retraining the model. This makes deepmufl quite fast and perhaps more attractive from a practical point of view. However, this comes with a risk of not being traceable, *i.e.*, a mutation on pre-trained .h5 model does not directly correspond to a line of source code for the user to inspect. In the Keras programs that we studied, this limitation was mitigated by the fact that the models with Sequential architecture were implemented using a well-understood structure and mapping layer numbers/identifiers in deepmufl's reports to source code was trivial. In a future work, with the help of simple auto-generated annotations, *e.g.*, for lexical scoping of the code snippet for model declaration, we will extend deepmufl to automatically map layer numbers/identifiers in its reports to actual lines of source code.

Humbatova *et al.* [27] argue about the importance of mutators emulating real DNN faults. We acknowledge that mutators emulating real faults would help generating more informative reports that would also give hints on how to fix the program. However, unlike mutation analysis, the main objective of an MBFL technique is to assign suspiciousness values to the program elements which can, in theory, be done using any kind of mutator, whether or not it makes sense from the standpoint of a developer. It is worth noting that the alternative design decision of using DeepCrime [27] as a mutation generation engine for deepmufl would result in finding more bugs than the current version of deepmufl, *e.g.*, bugs related to training hyper-parameters or training dataset, but such a design is expected to be impacted by the nondeterminacy inherent in training process and, given the fact that we do not employ any training data selection, would be significantly slower due to numerous re-training. Nevertheless, finding more bugs would be an incentive for exploring this alternative as a future work.

B. Test Case Splitter

Before we introduce this component, we need to clarify certain terminology.

Definition 1. *A data point in a testing dataset for a DNN model is defined to be a pair of the form (I, O), where $I \in \mathbb{R}^m$ and $O \in \mathbb{R}^n$, with m and n being positive integers. In this paper I is called test case, test input, or input, while O is called expected output or ground-truth value.*

Given a test dataset, test case splitter component applies the original model on each of the test cases for the data points and checks if the model output matches to the expected output. If the two outputs match, then the corresponding test case will be marked as *passing*, otherwise it will be marked as *failing*. This component also records the output produced by the original model to be used during impact analysis, described below.

Table 2: Summary of the 79 mutators implemented in deepmufl

Mutator Class	Description
MATH_WEIGHT	Add/subtract 1 to/from the weights of a given neuron and multiply/divide them to/by 2. Targets Dense and SimpleRNN layers.
MATH_WEIGHT_CONV	Add/subtract 1 to/from the weights of a convolution layer and multiply/divide them to/by 2. Targets subclasses of Conv, *i.e.*, Conv1D, Conv2D, *etc.*
MATH_ACT_WEIGHT	Add/subtract 1 to/from the activation weights of a (rolled) recurrent layer and multiply/divide them to/by 2. Targets SimpleRNN layers.
MATH_LSTM_IN_WEIGHT	Add/subtract 1 to/from the input weights of an LSTM layer and multiply/divide them to/by 2. Targets LSTM layers.
MATH_LSTM_FORGET_WEIGHT	Add/subtract 1 to/from the forget weights of an LSTM layer and multiply/divide them to/by 2. Targets LSTM layers.
MATH_LSTM_CELL_WEIGHT	Add/subtract 1 to/from the cell weights of an LSTM layer and multiply/divide them to/by 2. Targets LSTM layers.
MATH_LSTM_OUT_WEIGHT	Add/subtract 1 to/from the output weights of an LSTM layer and multiply/divide them to/by 2. Targets LSTM layers.
MATH_BIAS	Add/subtract 1 to/from the bias value of neuron and multiply/divide them to/by 2. Targets Dense and SimpleRNN layers.
DEL_LAYER	Deletes a Dense layer.
DUP_LAYER	Duplicates a Dense layer.
MATH_CONV_BIAS	Add/subtract 1 to/from the bias value of a convolution layer and multiply/divide them to/by 2. Target subclasses of Conv.
MATH_LSTM_IN_BIAS	Add/subtract 1 to/from the input bias of an LSTM layer and multiply/divide them to/by 2. Target LSTM layers.
MATH_LSTM_FORGET_BIAS	Add/subtract 1 to/from the forget bias of an LSTM layer and multiply/divide them to/by 2. Target LSTM layers.
MATH_LSTM_CELL_BIAS	Add/subtract 1 to/from the cell bias of an LSTM layer and multiply/divide them to/by 2. Target LSTM layers.
MATH_LSTM_OUT_BIAS	Add/subtract 1 to/from the output bias of an LSTM layer and multiply/divide them to/by 2. Target LSTM layers.
ACT_FUNC_REP	Replaces the activation of a neuron with a different one. Targets all layers with activation function in their configuration, *e.g.*, Dense, Conv2D, *etc.*
MATH_POOL_SZ	Increase/decrease pool size by 1, if applicable.
MATH_STRIDES	Increase/decrease strides by 1, if applicable.
MATH_KERNEL_SZ	Increase/decrease kernel size by 1, if applicable.
MATH_FILTERS	Increase/decrease filters by 1, if applicable.
PADDING_REP	Replace valid padding with same, *vice versa*. Targets subclasses of Conv.
REC_ACT_FUNC_REP	Replace the recurrent activation of a layer. Targets SimpleRNN layers.

C. Mutation Executor (Mutation Tester)

We start describing this component with a definition.

Definition 2. *A mutation execution matrix $\mathcal{E}$ is a $k \times l$ matrix, where k is the number of generated mutants, while l is the number of test cases. Each element $\mathcal{E}_{i,j}$ in the matrix is a member of the set $\{✓, ✗, \bigcirc\}$, wherein ✓ indicates that the ith mutant impacts j^{th} test case, whereas ✗ indicates that the mutant does not affect the test case. $\bigcirc$ denotes a nonviable mutant, i.e., a mutant that fails when loading or applying it on a test case. Such mutants might be generated, e.g., due to creating a shape error [32] after the mutation.*

Mutation executor component construct mutation execution matrix by applying each of the generated mutants (see §IV-A) on the failing and passing test cases to determine which of the test cases are impacted by which mutants. The impact of mutation on test cases is measured using two types of impacts, *i.e.*, *type 1* impact and *type 2* impact, defined below.

Definition 3. *Given a DNN model $\mathcal{M}$, its mutated version $\mathcal{M}'$, and a data point (I, O), we define the two types of impacts:*

- *Type 1: Mutant $\mathcal{M}'$ impacts the test case I if $\mathcal{M}(I) = O$ but $\mathcal{M}'(I) \neq O$, or $\mathcal{M}(I) \neq O$ but $\mathcal{M}'(I) = O$. In other words, type 1 impact tracks pass to fail and fail to pass test cases, à la MUSE [17].*
- *Type 2: Mutant $\mathcal{M}'$ impacts the test case I if $\mathcal{M}(I) \neq \mathcal{M}'(I)$.*

In this definition, $\mathcal{M}(I)$ or $\mathcal{M}'(I)$ denotes the operation of applying model $\mathcal{M}$ or $\mathcal{M}'$ on the test case I.

It is worth noting that checking for equality of two values can be tricky for regression models, as those models approximate the expected values. To work around this problem, deepmufl compares values obtained from regression models using a user-defined delta threshold, *i.e.*, the values are deemed equal if their absolute difference is no more than a threshold. By default, deepmufl uses a threshold of 0.001. This is the approach adopted by well-known testing frameworks for comparing floating-point values [54], [55]. Also, whether deepmufl uses type 1 or type 2 impact is a user preference and is determined alongside the threshold.

D. Suspiciousness Value Calculator

Armed with the above definitions, we can now give concrete definitions for the terms used in Eq. 1, 2, and 3, specialized to DNNs.

- Given a model element, *i.e.*, neuron or a layer, e, $M(e)$ is defined to be the set of all mutants generated by mutating e. These sets are produced by mutation generator process.
- Assuming that m is a mutant on the element e, $T_f(m, e)$ (or $T_p(m, e)$) is defined as the set of failing (or passing) test cases that are impacted in type 1 or type 2 fashion by m. More concretely, $T_p(m, e) = \{t \mid \mathcal{E}_{m,t} = ✓ \land t \text{ is passing}\}$, similarly $T_f(m, e) = \{t \mid \mathcal{E}_{m,t} = ✓ \land t \text{ is failing}\}$. These two sets are defined using a quite similar notation; the readers are encouraged to review Definitions 2 and 3 to avoid confusion.
- T_f (or T_p) is the set of originally failing (or passing) test cases. These sets are constructed by test case splitter.
- $F \rightsquigarrow P$ (or $P \rightsquigarrow F$), for a model element e, is defined as the set of originally failing (or passing) test cases that turned into passing (or failing) as a result of some mutation on e. More concretely, assuming the execution matrix $\mathcal{E}$ is constructed using type 1 impact, $F \rightsquigarrow P$ is defined as $\{t \mid t \text{ is failing} \land \exists m \in M(e) \cdot \mathcal{E}_{m,t} = ✓\}$. Similarly, $P \rightsquigarrow F$ is defined as $\{t \mid t \text{ is passing} \land \exists m \in M(e) \cdot \mathcal{E}_{m,t} = ✓\}$. In other words, these sets track all the failing/passing test cases that are *type 1 impacted* by some mutant of a given element. These two sets are defined using a quite similar notation; the readers are encouraged to review Definitions 2 and 3 to avoid confusion.

Having specialized definitions for the terms used in the fault localization formulas described earlier, we are now able to calculate suspiciousness values for the elements in a DNN model. Guided by the user preferences, deepmufl calculates all the values for $|T_f(m, e)|$, *etc.*, and plugs into the specified formula to calculate suspiciousness values for the elements. It is worth noting that if all the mutants generated for a given element are nonviable, MUSE formula (Eq. 3) and all the variants of Metallaxis (*e.g.*, Eq. 1), by definition, will return 0 as the suspiciousness value for the element. Nonviable mutants do not contribute toward localizing the bug, therefore they are considered *overhead* to the fault localization process. Fortunately, based on our observations in our dataset of buggy DNNs, nonviable mutants are rare.

Equivalent mutants are another source of overhead for deepmufl. Currently, we do not have any means of detecting equivalent mutants, but we argue that these mutants do not impact MBFL results, as they are equivalent to the original model and do not impact any passing or failing test cases.

V. SUPPORTED DNN BUGS

Due to the complex nature of DNN bugs, and MBFL itself, we do not hope to give a formal account of what types of DNN bugs deepmufl is capable of localizing. Instead, we attempt to provide as accurate description of the supported bugs as possible and discuss the way such bugs manifest in DNN programs. The discussion given in this section leverages the characterization of DNN bugs provided by previous research [7], [4], [6].

As we mentioned earlier, current version of deepmufl operates on pre-trained Keras `Sequential` models. This means that much of the information, such as training hyper-parameters and whether or not the input data is normalized, has already been stripped away from the input to deepmufl, and the current version of the technique is not capable of detecting any bug related to training process, *e.g.*, training data and hyper-parameters. Moreover, a pre-trained model does not contain bugs related to tensor shapes (as otherwise, the training would fail with shape errors), and since deepmufl does not receive the source code of the buggy model as input, bugs related to GPU usage and API misuse are also out of the reach of the technique, by definition. This leaves us with the so-called *model bugs* [7] the extent to which deepmufl is capable of localizing is explicated below. The four model bug sub-categories are represented with identifiers SC1, ..., SC4 in the rest of this paper for ease of reference.

- **SC1: Activation function**. These bugs are related to the use of wrong activation function in a layer. We observed that deepmufl detects this type of bugs and it also gives actionable, direct fixes.
- **SC2: Model type or properties**. These bugs include wrong weight initialization, wrong network architecture, wrong model for the task, *etc*. Through altering the weights and biases in layers, deepmufl detects weight/bias initialization bugs and pinpoint the location of the bug, but the bug report produced by the tool does not provide helpful information for fixing.
- **SC3: Layer properties**. These bugs include wrong filter/kernel/stride size, sub-optimal number of neurons in a layer, wrong input sample size, *etc*. deepmufl detects and pinpoints the bugs related to filter/kernel/stride size and sub-optimal number of neurons. We observed that, the former case sometimes produce non-viable mutants. In the cases where deepmufl produced viable mutants, effective MBFL takes place and it has been able to pinpoint the bug location and provide explanation on how to fix it. In the latter case, deepmufl was able to pinpoint the bug location, but the bug report does not give helpful information on how to fix the bugs in this sub-category.

- **SC4: Missing/redundant/wrong layer**. These bugs include missing/extra one dense layer, missing dropout layer, missing normalization layer, *etc*. By mutating the layers adjacent to the missing layer, or deleting the redundant layer, deepmufl detects and pinpoints the location of the missing/culprit layer, and in most of the cases, it provides useful information on how to fix such bugs.

By manually examining the bug descriptions provided by the programmers in our dataset of bugs, and also referring to the previous work on DNN bugs and root cause characterization [4], these bugs might manifest as low test accuracy/MSE, constant validation accuracy/MSE/loss during training, NaN validation accuracy/MSE/loss during training, dead nodes, vanishing/exploding gradient, and saturated activation.

At this point, we would like to emphasize that deepmufl is not intended to repair a model, so if a mutation happens to be the fix for the buggy model, the model has to be retrained from scratch so that correct weights and biases will be calculated.

VI. EVALUATION

We evaluate deepmufl and compare it to state-of-the-art static and dynamic DNN fault localization techniques, by investigating the following research questions (RQs).

- **RQ1 (Effectiveness)**:
  1) How does deepmufl compare to state-of-the-art tools in terms of the number of bugs detected?
  2) How many bugs does deepmufl detect from each sub-category of model bugs in our dataset and how does that compare to state-of-the-art tools?
  3) What are the overlap of detected bugs among deepmufl and other fault localization techniques?

- **RQ2 (Efficiency)**:
  1) What is the impact of mutation selection on the effectiveness and efficiency of deepmufl?
  2) How does deepmufl compare to state-of-the-art tools in terms of end-to-end fault localization time?

A. Dataset of DNN Bugs

To evaluate deepmufl and compare it to state-of-the-art DNN fault localization techniques, we queried StackOverflow Q&A website for posts about Keras that had at least one accepted answer. Details about the SQL query used to obtain the initial list of posts is available online [36]. The query resulted in 8,412 posts that we manually sieved through to find the programs with model bugs. Specifically, we kept the bugs that satisfied the following conditions.

- Implemented using `Sequential` API of Keras,
- The bug in the program was a *model bug* supported by deepmufl as described in §V, and
- The bug either had training datatset available in the post in some form (*e.g.*, hard-coded, clearly described in the body of the post, or a link to the actual data was provided) or we could see the described error using synthetic data obtained from `scikit-learn`'s dataset generation API.

This resulted in 102 bugs and we paired each bug with a fix obtained from the accepted answer to the question. We further added 7 bugs from DeepLocalize dataset [11] that are also coming from StackOverflow and we paired these bugs also with their fixes that are obtained from the most up-voted answer. Thus, we ended up with 109 bugs in total. To the best of our knowledge, this is the largest dataset of model bugs obtained from StackOverflow and it overlaps with the existing DNN bug datasets from previous research [12], [56]. Our bug dataset contains 85 classifiers (45 fully-connected DNNs, 29 CNNs, and 11 RNNs) and 24 regression models (19 fully-connected DNNs, 3 CNNs, and 2 RNNs). And each category has at least one example of model bugs. Therefore, we believe that our dataset is greatly representative of model bugs, *i.e.*, the bugs supported by deepmufl (and other tools that support this type of bugs), as we have examples of each sub-category of bug in various locations of the models for various regression and classification tasks.

After loading the training dataset for the bugs, we fitted the buggy models three times and stored them in .h5 file format separately. The repetition was conducted to take randomness in training into account. Randomness in data generation was mitigated by using deterministic random seeds. For fault localization purposes, we used the test dataset, and if it was not available, we used the training dataset itself. When we had to use synthesized data points, we deterministically splitted the generated set of data into training and testing datasets.

B. Baseline Approaches and Measures of Effectiveness

In RQ1 and RQ2, we compare five different configurations of deepmufl to recent static and dynamic DNN fault localization tools. The five configurations of deepmufl are as follows.

- **Metallaxis** [30]: In this setting, we use the Metallaxis formula to calculate suspiciousness values of model elements. Metallaxis, by default, uses SBI [46] to calculate suspiciousness values for individual mutants. A recent study [57] provides empirical evidence on the superiority of Ochiai [47] over SBI when used within Metallaxis formula. Thus, we considered the following four combinations: *type 1 impact*: (1) SBI formula (*i.e.*, Eq. 1); (2) Ochiai formula (*i.e.*, Eq. 2), and *type 2 impact*: (3) SBI formula (*i.e.*, Eq. 1); (4) Ochiai formula (*i.e.*, Eq. 2).
- **MUSE** [17]: We used the default formula of MUSE to calculate the suspiciousness of model elements. For this, only type 1 impact is considered, as the heuristics behind MUSE are defined based on type 1 impact.

Our technique follows a more traditional way of reporting root causes for the bugs [30], [17], [57], [19], [22], [21], [15], in that it reports a list of potential root causes ranked based on the likelihood of being responsible for the bug. This allows the users find the bugs faster and spend less time reading through the fault localization report, which in turn increases practicality of the technique [58]. We have used top-N, with $N = 1$, metric to measure the effectiveness of deepmufl in RQ1 and RQ2. Specifically, if the numbers of any of the buggy layers of the bug appeared in the first place in the output of deepmufl,

Table 3: Effectiveness of different deepmufl configurations and four other tools in detecting bugs from four sub-categories of model bugs

deepmufl configuration / tool	SC 1	SC 2	SC 3	SC 4	Total (detected)
Metallaxis SBI + Type 1	31	2	6	3	42
Metallaxis Ochiai + Type 1	36	2	7	2	47
Metallaxis SBI + Type 2	18	2	4	2	26
Metallaxis Ochiai + Type 2	29	2	4	2	37
MUSE	41	3	6	3	53
Neuralint	15	1	4	1	21
DeepLocalize	21	0	4	1	26
DeepDiagnosis	22	2	5	1	30
UMLAUT	18	1	6	0	25
Total (entire dataset)	80	4	17	8	

we reported it as *detected*, otherwise we marked the bug as *not-detected*. We emphasize that top-1 metric gives a strong evidence on the effectiveness of deepmufl, as the developers usually only inspect top-ranked elements, *e.g.*, over 70% of the developers only check top-5 ranked elements [59].

Our selection criteria for the studied fault localization techniques are: (1) availability; (2) reproducibility of the results reported in the original papers, so as to have a level of confidence on the correctness of the results reported here; and (3) support for *model bugs* in our dataset, so that we can make a meaningful comparison to deepmufl. Below we give a brief description of each of the selected tools, why we believe they support model bugs, and how we have interpreted their outputs in our experiments, *i.e.*, when we regard a bug being detected by the tool.

1) Neuralint: A static fault localization tool that uses 23 rules to detect faults and design inefficiencies in the model. Each rule is associated with a set of rules of thumb to fix the bug that are shown to the user in case the precondition for any of the rules are satisfied. The five rules described in Section 4.2.1 of the paper target model bugs. Neuralint produces outputs of the form [Layer $L ==> MSG$]*, where L is the suspicious layer number, and MSG is a description of the detected issue and/or suggestion on how to fix the problem. A bug is deemed *detected* by this tool if it is located in the layer mentioned in the output message or the messages describe any of the root causes of the bug.

2) DeepLocalize: A dynamic fault localization technique that detects numerical errors during model training. One of three rules described in Section III.D of the paper checks model bugs related to wrong activation function. DeepLocalize produces a single message of the form Batch B Layer L : MSG, where B is the batch number wherein the symptom is detected and L and MSG are the same as we described for Neuralint. A bug is deemed *detected* if it is located in the layer mentioned in the output message or the message describes any of the root causes of the bug.

3) DeepDiagnosis: A tool similar to DeepLocalize, but with more bug pattern rules and a decision procedure to give actionable fix suggestions to the users based on the observations. All 8 rules in Table 2 of the paper monitor the symptoms of model bugs. Similar to DeepLocalize, DeepDiagnosis produces a single message of the form Batch B Layer L : MSG_1 [OR MSG_2], where B and L are the same as described in DeepLocalize and MSG_1 and

MSG_2 are two alternative solutions that the tool might suggest to fix the detected problem. A bug is deemed *detected* if it is located in the layer mentioned in the output message or the message describes any of the root causes of the bug.

4) UMLAUT: A hybrid, *i.e.*, a combination of static and dynamic, technique that works by applying heuristic static checks on, and injecting dynamic checks in, the program, parameters, model structure, and model behavior. Violated checks raise error flags which are propagated to a web-based interface that uses visualizations, tutorial explanations, and code snippets to help users find and fix detected errors in their code. All three rules described in Section 5.2 of the paper target model bugs. The tool generates outputs of the form $[< MSG_1 > \cdots < MSG_m >]^*$, where $m > 0$ and MSG_i is a description of the problem detected by the tool. A bug is deemed *detected* if any of the messages match the fix prescribed by the ground-truth.

C. Results

To answer RQ1, we ran deepmufl (using its five configurations) and four other tools on the 109 bugs in our benchmark. We refer the reader to the repository [36] for the raw data about which bug is detected by which tool, and here we describe the summaries and provide insights.

At top-1, deepmufl detects 42, 47, 26, 37, and 53 bugs using its Metallaxis SBI + Type 1, Metallaxis Ochiai + Type 1, Metallaxis SBI + Type 2, Metallaxis Ochiai + Type 2, and MUSE, respectively, configurations. Meanwhile Neuralint, DeepLocalize, DeepDiagnosis, and UMLAUT detect 21, 26, 30, and 25, respectively, bugs. Therefore, as far as the number of bugs detected by each technique is concerned, MUSE configuration of deepmufl is the most effective configuration of deepmufl, significantly outperforming studied techniques, and Metallaxis Ochiai + Type 2 is the least effective one, outperformed by DeepDiagnosis. An empirical study [57], which uses a specific dataset of traditional buggy programs, concludes that Metallaxis Ochiai + Type 2 is the most effective configuration for MBFL. Meanwhile, our results for DNNs corroborates the theoretical results by Shin and Bae [60], *i.e.*, we provide empirical evidence that in the context of DNNs MUSE is the most effective MBFL approach.

Table 3 reports more details and insights on the numbers discussed above. Specifically, it reports the number of bugs detected by each configuration of deepmufl an four other studied tools from each sub-category of model bugs present in our dataset of bugs. As we can see from the upper half of the table, MUSE is most effective in detecting bugs related to activation function (SC1), bugs related to model type/properties (SC2), and wrong/redundant/missing layer (SC4), while Metallaxis Ochiai + Type 1 configuration outperforms other configurations in detecting bugs related to layer properties (SC3). Similarly, from bottom half of the table, we can see that other tools are also quite effective in detecting bugs related to activation function, with DeepDiagnosis being the most effective one among others. We can also observe that UMLAUT has been the most effective tool in detecting bugs

Table 4: The fraction of bugs detected by each deepmufl configuration that are also detected by the other four tools

	Neuralint	DeepLocalize	DeepDiagnosis	UMLAUT	Combined
Metallaxis SBI + Type 1	23.81% (10)	19.05% (8)	21.43% (9)	19.05% (8)	59.52% (25)
Metallaxis Ochiai + Type 1	27.66% (13)	25.53% (12)	21.28% (10)	17.02% (8)	57.45% (27)
Metallaxis SBI + Type 2	15.38% (4)	15.38% (4)	11.54% (3)	15.38% (4)	46.15% (12)
Metallaxis Ochiai + Type 2	13.51% (5)	18.92% (7)	21.62% (8)	16.22% (6)	48.65% (18)
MUSE	22.64% (12)	22.64% (12)	28.3% (15)	24.53% (13)	60.38% (32)

related to layer properties. As we can see, MUSE configuration of deepmufl is consistently more effective than other tools across all bug sub-categories.

Table 4 provides further insights on the overlap of bugs detected by each variant of deepmufl and those detected by the other four tools. Each value in row r and column c of this table, where $2 \leq r \leq 5$ and $2 \leq c \leq 6$, denotes the percentage of bugs detected by the deepmufl variant corresponding to row r and tool corresponding to column c. The values inside the parenthesis are the actual number of bugs. For example, 8 out of 42, *i.e.*, 19.05%, of the bugs detected by Metallaxis SBI + Type 1 configuration of deepmufl are *also* detected by DeepLocalize. The last column of the table reports same statistics, except for all four of the studied tools combined. As we can see from the table, 60.38% of the bugs detected by MUSE configuration of deepmufl are already detected by one of the four tools, yet it detects 21 (=53-32) bugs that are not detected by any other tools. This is because deepmufl approaches fault localization problem from a fundamentally different aspect giving it more flexibility. Specifically, instead of looking for conditions that trigger a set of hard-coded rules, indicating bug patterns, deepmufl breaks the model using a set of mutators to observe how different mutation impact the model behavior. Then by leveraging the heuristics underlying traditional MBFL techniques, it performs fault localization using the observed impacts on the model behavior. Listing 2 shows an example of a model bug that only deepmufl can detect.

```
1  # load and split the dataset
2  # ...
3  model = Sequential()
4  model.add(Dense(4, input_dim=2, activation='relu'))
5  model.add(Dense(1, activation='relu'))
6  model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['MSE'])
7  model.fit(X, Y, epochs=500, batch_size=10)
```

Listing 2: Bug 48251943 in our dataset

The problem with this regression model is that it does not output negative values, and given the fact that the dataset contains negative target values, the model achieves high MSE. Since this model does not result in any numerical errors during training, DeepLocalize does not issue any warning messages, and since MSE decreases and the model does not show any sign of erratic behavior DeepDiagnosis does not detect the bug. UMLAUT's messages instruct adding softmax layer and checking validation accuracy which is clearly not related to the problem, because the bug is fixed by changing the activation function of the last layer to tanh and normalizing the output values. Lastly, Neuralint issues an error message regarding incorrect loss function which also seems to be a false positive.

To answer RQ2, we ran `deepmufl` and the other four tools on a Dell workstation with Intel(R) Xeon(R) Gold 6138 CPU at 2.00 GHz, 330 GB RAM, 128 GB RAM disk, and Ubuntu 18.04.1 LTS and measured the time needed for model training as well as the MBFL process to complete. We repeated this process four times, and in each round of `deepmufl`'s execution, we randomly selected 100% (*i.e.*, no selection), 75%, 50%, and 25% of the generated mutants for testing. Random mutation selection is a common method for reducing the overhead of mutation analysis [61], [35]. During random selection, we made sure that each layer receives at least one mutants, so that we do not mask any bug. The last row in Table 5 reports the average timing (of 3 runs) of MBFL in each round of mutation selection. The table also reports the impact of mutation selection on the number of bugs detected by each configuration of `deepmufl`. As we can see, in MUSE configuration of `deepmufl`, by using 50% of the mutants, one can halve the execution time and still detect 92.45% of the previously detected bugs. Therefore, mutation selection can be used as an effective way for curtailing MBFL time in DNNs.

For a fair comparison of `deepmufl` to state-of-the-art fault localization tools in terms of efficiency, we need to take into account the fact that `deepmufl` requires a pre-trained model as its input. Thus, as far as the end-to-end fault localization time from an end-user's perspective is concerned, we want to take into consideration the time needed to train the input model in addition the time needed to run `deepmufl`. With training time taken into account, `deepmufl` takes, on average, 1492.48, 1714.63, 1958.35, and 2192.4 seconds when we select 25%, 50%, 75%, and 100% of the generated mutants, respectively. We also emphasize that the time for DeepLocalize and DeepDiagnosis varied based on whether or not they found the bug. Given the fact that a user could terminate the fault localization process after a few epochs when they lose hope in finding bugs with these two tools, we report two average measurements for DeepLocalize and DeepDiagnosis: (1) average time irrespective of the fact that the tools succeed in finding the bug; (2) average time if the tools successfully finds the bug. Unlike these two tools, the time for Neuralint and UMLAUT does not change based on the fact that they detect a bug or not. DeepLocalize takes on average 1244.09 seconds and it takes on average 57.29 seconds when the tool successfully finds the bug. These numbers for DeepDiagnosis are 1510.71 and 11.05 seconds, respectively. Meanwhile, Neuralint and UMLAUT take on average 2.87 seconds and 1302.61 seconds to perform fault localization.

D. Discussion

It is important to note that while `deepmufl` outperforms state-of-the-art techniques in terms of the number of bugs detected in our dataset, it is not meant to replace them. Our dataset only covers a specific type of bugs, *i.e.*, model bugs, while other studied techniques push the envelope by detecting bugs related to factors like learning rate and training data normalization, which are currently outside of `deepmufl`'s reach. We observed that combining all the techniques results

Table 5: The impact of mutation selection on the effectiveness and execution time of `deepmufl`

	Selected mutants			
	25%	50%	75%	100%
Metallaxis SBI + Type 1	37	41	42	42
Metallaxis Ochiai + Type 1	40	46	47	47
Metallaxis SBI + Type 2	25	26	26	26
Metallaxis Ochiai + Type 2	34	37	37	37
MUSE	42	49	51	53
Time (s)	340.58	562.72	806.45	1,040.49

in detecting 87 of the bugs in our dataset; exploring ways to combine various fault localization approaches by picking the right tool based on the characteristics of the bug is an interesting topic for future research. Moreover, depending on the applications and resource constraints, a user might prefer one tool over another. For example, although Neuralint might be limited by its static nature, *e.g.*, it might not be able analyze models that use complex computed values and objects in their construction, it takes only few seconds for the tool to conduct fault localization. Thus, in some applications, *e.g.*, online integration with IDEs, approaches like that of Neuralint might be the best choice.

A major source of overhead in an MBFL technique is related to the sheer number of mutants that the technique generates and tests [62], [61]. Sufficient mutator selection [63] is referred to the process of selecting a subset of mutators that achieve the same (or similar) effect, *i.e.*, same or similar mutation score and same or similar number of detected bugs, but with smaller number of mutants generated and tested. For the mutators of Table 2, so far, we have not conducted any analysis on which mutators might be redundant, as a reliable mutator selection requires a larger dataset that we currently lack. We postpone this study as a future work.

Combining fault localization tools can be conducted with the goal of improving efficiency. We see the opportunity in building faster, yet more effective, fault localization tools by predicting the likely right tool upfront for a given model or running tools one by one and moving on to the next tool if we have a level of confidence that the tool will not find the bug. We postpone this study for a future work.

Lastly, we would like to emphasize that comparisons to the above-mentioned techniques in a dataset of bugs that `deepmufl` supports is fair, as the other tools are also designed to detect bugs in the category of model bugs. However, making these tools to perform better than this, would require augmenting their current rule-base with numerous new rules, yet adding new rules comes with the obligation of justifying the generality and rationale behind them, which might be a quite difficult undertaking. `deepmufl`, on the other hand, approaches the fault localization problem differently, allowing for more flexibility without the need for hard-coded rules.

VII. THREATS TO VALIDITY

As with most empirical evaluations, we do not have a working definition of representative sample of DNN bugs, but we made efforts to ensure that the bugs we used in the evaluation is as representative as possible by making sure

that our dataset has diverse examples of bugs from each sub-category of model bugs.

Many of the bugs obtained from StackOverflow did not come with accompanying training datasets. To address this issue, we utilized the dataset generation API provided by `scikit-learn` [64] to generate synthetic datasets for regression or classification tasks. We ensured that the errors described in each StackOverflow post would manifest when using the synthesized data points and that applying the fix suggested in the accepted response post would eliminate the bug. However, it is possible that this change to the training process may introduce new unknown bugs. To mitigate this risk, we have made our bug benchmark publicly available [36].

Another potential threat to the validity of our results is the possibility of bugs in the construction of deepmufl itself, which could lead to incorrect bug localization. To mitigate this, we make the source code of deepmufl publicly available for other researchers to review and validate the tool.

Another threat to the validity of our results is the potential impact of external factors, such as the stochastic nature of the training process and the synthesized training/testing datasets, as well as system load, on our measurements. To address this, besides using deterministic seeds for dataset generation and splitting, we repeated our experiments with deepmufl three times. Similarly, we also ran other dynamic tools three times to ensure that their results were not affected by randomness during training. We did not observe any differences in effectiveness between the rounds for either deepmufl or the other studied techniques. Additionally, we repeated the time measurements for each round, and reported the average timing, to ensure that our time measurements were not affected by system load. Furthermore, judging whether or not any of the tools detect a bug requires manual analysis of textual description of the bugs and matching it to the tools; output messages which might be subject to bias. To mitigate this bias, we have made the output messages by the tools available for other researchers [36].

Lastly, deepmufl uses a threshold parameter to compare floating-point values (see §IV-C). In our experiments, we used the default value of 0.001 and ensured that smaller threshold values yield the same results.

VIII. Related Work

Neuralint [12] uses *graph transformations* [65] to abstract away unnecessary details in the model and check the bug patterns directly on the graph. While Neuralint is orders of magnitude faster than deepmufl, it proved to be less effective than deepmufl in our dataset.

DeepLocalize [11] and DeepDiagnosis [8] intercept the training process looking for known bug patterns such as numerical errors. DeepDiagnosis pushes the envelope by implementing a decision tree that gives actionable fix suggestions based on the detected symptoms. A closely related technique, UMLAUT [34], works by applying heuristic static checks on, and injecting dynamic checks in, various parts of the DNN

program. deepmufl outperforms DeepLocalize, DeepDiagnosis, and UMLAUT in terms of the number of bugs detected.

DeepFD [66] is a recent learning-based fault localization technique which frames the fault localization as a learning problem. MODE [25] and DeepFault [26] implement white-box DNN testing technique which utilizes suspiciousness values obtained *via* an implementation of spectrum-based fault localization to increase the hit spectrum of neurons and identify suspicious neurons whose weights have not been calibrated correctly and thus are considered responsible for inadequate DNN performance. MODE was not publicly available, but DeepFault was, but unfortunately it was hard-coded to the examples shipped with its replication package, so we could not make the tool work without making substantial modifications to it, not to mention that these techniques work best on ReLU-based networks and applying them on most of the bugs in our dataset would not make much sense.

Other related works are as follows. PAFL [67] operates on RNN models by converting such models into probabilistic finite automata (PFAs) and localize faulty sequences of state transitions on PFAs. Sun *et al.* [68] propose DeepCover, which uses a variant of spectrum-based fault localization for DNN explainability.

IX. Conclusion

This paper revisits mutation-based fault localization in the context of DNN and presents a novel DNN fault localization technique, named deepmufl. The technique is based on the idea of mutating a pre-trained DNN model and calculating suspiciousness values according to Metallaxis and MUSE approaches, Ochiai and SBI formulas, and two types of impacts of mutations on the results of test data points. deepmufl is compared to state-of-the-art static and dynamic fault localization systems [11], [8], [34], [12] on a benchmark of 109 model bugs. In this benchmark, while deepmufl is slower than the other tools, it proved to be almost two times more effective than them in terms of the total number of bugs detected and it detects 21 bugs that none of the studied tools were able to detect. We further studied the impact of mutation selection on fault localization time. We observed that we can halve the time taken to perform fault localization by deepmufl, while losing only 7.55% of the previously detected bugs.

REFERENCES

[1] *IEEE Standard Classification for Software Anomalies*, 2010.

[2] A. McPeak, "What's the true cost of a software bug?" https://smartbear.com/blog/software-bug-cost/, 2017, accessed 08/10/23.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *ESEC/FSE*, 2019, pp. 510–520.

[5] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *ICSE*, 2020, pp. 1135–1146.

[6] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *ISSTA*, 2018, pp. 129–140.

[7] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020, pp. 1110–1121.

[8] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *ICSE*. IEEE, 2022, pp. 561–572.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *SOSP*, 2017, pp. 1–18.

[10] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *ICSE*, 2019, pp. 1039–1049.

[11] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: fault localization for deep neural networks," in *ICSE*, 2021, pp. 251–262.

[12] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *TOSEM*, vol. 31, no. 1, pp. 1–27, 2021.

[13] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu, "Nn repair: Constraint-based repair of neural network classifiers," in *CAV*, 2021, pp. 3–25.

[14] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *ICSE*, 2021, pp. 359–371.

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *TSE*, vol. 42, no. 8, pp. 707–740, 2016.

[16] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *ICST*, 2012, pp. 691–700.

[17] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014, pp. 153–162.

[18] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, pp. 34–41, 1978.

[19] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[20] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Bug Localization with AMPLE," in *ISAADD*, 2005, pp. 99–104.

[21] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002, pp. 467–477.

[22] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *TOSEM*, vol. 20, no. 3, pp. 1–32, 2011.

[23] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *SBSE*, 2012, pp. 244–258.

[24] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *SBSE*, 2013, pp. 224–238.

[25] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *ESEC/FSE*, 2018, pp. 175–186.

[26] H. F. Eniser, S. Gerasimou, and A. Sen, "Deepfault: Fault localization for deep neural networks," in *FASE*, 2019, pp. 171–191.

[27] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: mutation testing of deep learning systems based on real faults," in *ISSTA*, 2021, pp. 67–78.

[28] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *ASE*, 2019, pp. 1158–1161.

[29] 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 *ISSRE*, 2018, pp. 100–111.

[30] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *STVR*, vol. 25, no. 5-7, pp. 605–628, 2015.

[31] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[32] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016, p. 265–283.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019, pp. 8024–8035.

[34] E. Schoop, F. Huang, and B. Hartmann, "Umlaut: Debugging deep learning programs using program structure and model behavior," in *CHI*, 2021, pp. 1–16.

[35] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *JSS*, pp. 185–196, 1995.

[36] A. Ghanbari, D.-G. Thomas, M. A. Arshad, and H. Rajan, "Mutation-based fault localization of deep neural networks," https://github.com/ali-ghanbari/deepmufl-ase-2023, 2023.

[37] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*, 2005, pp. 402–411.

[38] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, 2019, vol. 112, pp. 275–378.

[39] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, 2010, pp. 65–74.

[40] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019, pp. 19–30.

[41] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *ESE*, pp. 783–812, 2015.

[42] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *IWSBST*, 2016, pp. 45–54.

[43] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test case prioritization techniques," *STVR*, p. e1695, 2019.

[44] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller, "Inferring loop invariants by mutation, dynamic analysis, and static checking," *TSE*, pp. 1019–1037, 2015.

[45] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? falsification-driven verification (t)," in *ASE*, 2015, pp. 737–748.

[46] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[47] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC*, 2006, pp. 39–46.

[48] Wikipedia contributors, "Hierarchical data format — Wikipedia, the free encyclopedia," 2022, accessed 08/10/23.

[49] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *ISSTA*, 2016, pp. 449–452.

[50] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[51] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *ASE*, 2011, pp. 612–615.

[52] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *STVR*, pp. 97–133, 2005.

[53] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *ESEC/FSE*, 2009, pp. 297–298.

[54] "Junit," http://junit.org/, 2019, accessed 08/10/23.

[55] "Testng documentation," https://testng.org/doc/documentation-main.html, 2017, accessed 08/10/23.

[56] M. M. Morovati, A. Nikanjam, F. Khomh, Z. Ming *et al.*, "Bugs in machine learning-based systems: A faultload benchmark," *arXiv*, 2022.

[57] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[58] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011, pp. 199–209.

[59] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*, 2016, pp. 165–176.

[60] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *ICST*, 2016, pp. 299–308.

[61] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.

[62] J. Zhang, "Scalability studies on selective mutation testing," in *ICSE*, vol. 2, 2015, pp. 851–854.

[63] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software: Practice and Experience*, vol. 26, no. 2, pp. 165–176, 1996.

[64] scikit-learn Contributors, "`scikit-learn`: Machine learning in python," 2020, accessed 08/10/23. [Online]. Available: https://scikit-learn.org/stable/

[65] R. Heckel, "Graph transformation in a nutshell," *ENTCS*, vol. 148, no. 1, pp. 187–198, 2006.

[66] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, "Deepfd: Automated fault diagnosis and localization for deep learning programs," in *ICSE*, 2022, p. 573–585.

[67] Y. Ishimoto, M. Kondo, N. Ubayashi, and Y. Kamei, "Pafl: Probabilistic automaton-based fault localization for recurrent neural networks," *IST*, vol. 155, p. 107117, 2023.

[68] Y. Sun, H. Chockler, X. Huang, and D. Kroening, "Explaining image classifiers using statistical fault localization," in *ECCV*, 2020, pp. 391–406.