Rohan Arora (UNI: ra3091)
Mar 4, 2024
COMS 6156

A Review of Fault Localization Techniques for Deep Neural Networks

Abstract – Deep neural networks (DNNs) are susceptible to a variety of bugs. Given their increasing popularity, localization of faults in DNNs is important for ensuring model safety and reliability. This paper categorizes recent fault localization techniques into three high level categories: those which localize (1) faults during system interactions, (2) data or configuration faults, and (3) model structure or input faults. Throughout this paper, an understanding of each technique is realized, including their advantages and drawbacks. Finally, an experiment is performed which shows that deepmufl remains effective at localizing bugs when mutation selection is used to halve execution time, with only a 9.57% reduction in bugs localized.

1 Introduction

Given the increasing utilization of deep neural networks (DNNs) and their usage in a variety of applications, it is important that these models are safe and reliable [1]. DNNs are susceptible to a variety of bugs which may arise during system interactions, relate to data or configuration faults, or relate to model structure or inputs [8]. This paper reviews recent techniques that localize faults in DNNs. As part of this review, fault localization techniques are categorized by the types of bugs each is able to identify, and comparisons in ability and efficiency are made. Finally, an experiment is described which attempts to replicate key findings from a recent fault localization technique, with areas of future research proposed.

What the Reader is Expected to Learn: After reading this paper, the reader should have an understanding of the various faults that may arise in DNNs. In addition, the reader should be familiar with recent fault localization techniques, including the types of bugs each is able to localize and how each technique operates. The reader should also be able to compare the efficiency of different methods. Finally, the reader should understand how to replicate key results from the paper by Ghanbari *et al.* [1], and potentially have some ideas on how to extend their work.

2 Background

2.1 DNN Faults

Compared to traditional programs, which do not have a data-driven component, bugs in DNNs are considered to be more difficult to detect [1]. This is primarily due to DNN performance relying on the characteristics and quality of training data, in addition to various hyperparameters that control the training process [1], [6], [8].

Recent DNN fault localization techniques are able to detect faults from the following categories: (1) *model bugs* [1], [2], [3], [4], [5], [6], [7], [10], [11], (2) *hyperparameter / training configuration bugs* [3], [4], [6], [7], [10], (3) *dataset bugs* [3], [6], [7], [12], (4) *API misuse* [2], (5) *GPU / hardware issues* [3], and (6) *tensor / input bugs* [2], [3]. These categories generally follow those from the taxonomy of faults in deep learning systems proposed by Humbatova *et al.* [8], and are briefly described in sections 2.1.1 through 2.1.6.

Although issues with model inaccuracy and out-of-distribution detection could arise from the faults described in this section, model inaccuracy and out-of-distribution detection issues should be considered as distinct problems, since they may exist even in the absence of the following faults.

2.1.1 Model Bugs

Model bugs broadly refers to faults which relate to the structure or properties of a deep learning model [8]. This category includes faults which:

- *Impact the model as a whole*. For example, usage of an incorrect model type (e.g., recurrent network used in place of a convolutional network), or usage of a network structure which is too complex or too simple in terms of the number of layers used [8].
- *Impact particular layers of a network.* This sub-category includes faults that are local to a specific layer. For example, a missing/redundant/wrong layer, incorrect activation function, and incorrect layer properties (e.g., incorrect input/output shape or number of neurons utilized) [8].

2.1.2 Hyperparameter / Training Configuration Bugs

This category of faults includes the usage of suboptimal hyperparameter settings which may impact training time or performance of a deep learning model [8]. The most commonly reported hyperparameters causing bugs include learning rate, batch size, and number of epochs [8]. Other hyperparameters such as the optimizer and momentum of the optimizer may cause faults in deep learning models, and are included in this category. However, network architecture hyperparameters such as the number of layers, number of neurons per layer, activation function, and other layer properties are excluded, since these are considered to be model bugs.

In addition, other configuration settings which impact training are included in this category, such as the suboptimal selection of the loss function (as opposed to the selection of an incompatible loss function, which would be considered as API misuse). The rationale behind grouping these types of bugs together is that they are either selected before or updated during the training process. In addition, hyperparameter / training configuration bugs cannot be detected solely from the trained model, at least based on the latest fault localization methods in this review, whereas model bugs may be detected from the trained model itself [1].

2.1.3 Dataset Bugs

Faults that relate to the data utilized for training models are considered in this category. These faults include issues with preprocessing (e.g., normalization required but not utilized), insufficient quantity of training data, imbalanced dataset, incorrectly labeled training data, and low quality training dataset [8].

2.1.4 API Misuse

Misuse of a deep learning framework or API occurs when logic is used which does not conform with how the developers of the framework intended for it to be utilized [8]. Examples of API misuse include: incompatible loss function (e.g., binary cross-entropy loss function used with softmax as the final layer activation function), and optimizer undefined or disconnected from the loss and learnable parameters [2].

2.1.5 GPU / Hardware Issues

GPU and hardware issues tend to be quite specific when they arise in deep learning tasks [8]. Examples of these faults include parallelism failures, incorrect GPU device references, state sharing errors in subprocesses, faulty GPU data transfers, and suboptimal channel ordering (for deep learning tasks with image inputs) [3], [8].

2.1.6 Tensor / Input Bugs

Faults in this category involve issues with the format, type, or shape of data used as input to a layer or method [8]. Faults related to tensor shapes are also included here, which arise when an operation is attempted on a tensor with an unexpected shape, or a single tensor is defined with the incorrect shape [2], [8].

2.2 Mutation Analysis

Mutation analysis is a technique which creates a set of program variants (*mutants*) by systematically mutating each program element [1]. In traditional mutation analysis, these mutants are run against a test suite so that the quality of the test suite can be determined [1]. Test suite quality is calculated through a mutation score, which conveys how effective the test suite is at detecting mutations introduced to a program [1].

While mutation analysis was originally used for traditional programs, it has recently been applied to deep learning contexts for fault localization [1], [4], [10], [12], and assessment of test suite quality [9]. For deep learning fault localization approaches which utilize mutation analysis, the process varies depending on the specific use case. However, the common theme remains of systematically mutating some aspect of the deep learning process (e.g., the model itself or the training dataset) to achieve fault localization.

It is worth pointing out that while the test suite in a traditional program may typically be a set of test cases written by an engineer, the test suite in a deep learning context is likely to be the test dataset, which is generally not produced by an engineer [1].

2.3 Spectrum-based Fault Localization

Localization of faults may be achieved through a technique known as spectrum-based fault localization [1], [11]. Similar to traditional mutation analysis, spectrum-based fault localization utilizes the test suite [11]. However, instead of mutating the original program, spectrum-based fault localization identifies faulty elements by identifying which pieces of code are executed by passing and failing test cases, with the idea being that elements executed by failing test cases are more likely to be faulty than those executed by passing test cases [11]. Recently, several techniques have extended spectrum-based fault localization to apply to deep learning contexts [1], [11], [12].

2.4 Static & Dynamic DNN Fault Localization

Each DNN fault localization technique in this review achieves fault localization through *static* [2], [5], *dynamic* [1], [4], [6], [7], [10], [11], [12], or a combination of static and dynamic approaches [3]. While static approaches localize faults without actually running the model by examining model properties or code, dynamic approaches involve running the model (or variants of the model) directly to observe behavior for fault localization purposes.

3 Fault Detection During System Interactions

The performance of DNNs relies heavily on the execution environment, including the APIs of supporting frameworks and underlying hardware (e.g., GPUs) [8]. Bugs related to how DNNs interact with these systems are a distinct class of faults that often manifest during runtime as crashes or errors [8]. This section focuses on recent fault localization techniques that analyze how DNNs interact with their execution environments and localize faults related to API misuse and GPU / hardware issues, previously described in sections 2.1.4 through 2.1.5.

`NeuraLint` is one such technique which is able to localize API misuse faults [2]. As presented in the paper by Nikanjam *et al.*, `NeuraLint` supports feedforward neural networks, including convolutional networks, but could possibly be extended in the future to support other architectures, such as recurrent neural networks [2]. `NeuraLint` functions by constructing a meta-model of the deep learning model under analysis from its source code (importantly, not the trained model itself) [2]. This meta-model is represented as a graph, and various graph transformations are utilized to identify bug patterns directly on the meta-model graph [1], [2]. These graph transformations essentially execute a set of verification rules which are applied to the meta-model graphs, and any violations of these rules suggest a detected fault [2]. Since the model itself is not trained or executed, `NeuraLint` is considered a static fault localization approach, which offers the advantage of efficiency [1], [2]. `NeuraLint` takes approximately 3.11 seconds to execute on a 38 layer deep learning program based on the paper by Nikanjam *et al.* [2], and only 2.87 seconds on average based on the paper by Ghanbari *et al.* [1]. The execution time results from Nikanjam *et al.* are shown in Table 1 (Appendix C), and those from Ghanbari *et al.* are shown in Table 1 (Appendix B).

Out of all recent fault localization techniques under review, `NeuraLint` is the only technique which is able to localize API misuse faults. `NeuraLint` is able to leverage its graph-based verification approach to localize API misuse faults, such as:

- Incorrectly defined loss or loss which is incompatible with the activation function of the final layer [2].
- Incorrectly defined optimizer or disconnected optimizer from the computational graph [2].
- Incorrect initialization of learnable parameters, which should be initialized once at the start of training [2].
- Failure to re-initialize gradients after each training iteration [2].
- Failure to solve loss minimization problem iteratively with continuously updated parameters [2].

A second technique which is able to detect faults during system interactions, specifically the localization of GPU / hardware issues, is `UMLAUT` [3]. Based on the original paper by Schoop *et al.*, `UMLAUT` supports classification or regression outputs with image or sparse text inputs, with the possibility to be extended to other data types [3]. However, based on the papers by Wardat *et al.* and *Cao et al.*, their analyses show that `UMLAUT` only supports classification models with softmax as the final layer activation function, and will otherwise report false alarms [4], [6]. This is further substantiated by the original paper on `UMLAUT`, which states that since checks on the model are implemented with heuristics, false positives (as well as false negatives) may arise [3].

`UMLAUT` operates on a DNN program by invoking heuristic static checks and injecting dynamic checks into the training routine, which monitor parameters, model structure, and model behavior [1], [3]. Any heuristics that are violated are then shown in a web interface, along with resources to localize and resolve bugs in the model [3]. Out of all recent fault localization techniques under review, `UMLAUT` is the only technique which uses a combination of static and dynamic checks. `UMLAUT` is also the only technique out of those under review which is able to

localize GPU / hardware issues. When operating on models with images as input data, UMLAUT will report an error if the channel ordering is not optimal for the hardware being used [3].

While both NeuraLint and UMLAUT are able to detect faults during system interactions, the types of faults, in addition to the techniques employed to localize them are quite different. NeuraLint utilizes static checks to localize API misuse faults via graph transformations [2], while UMLAUT utilizes dynamic checks injected into the training process to detect GPU / hardware issues [3]. NeuraLint is able to support feedforward, convolutional, and (could be extended to support) recurrent neural networks [2], which is similar to the model types that are supported by UMLAUT [3]. However, UMLAUT primarily works with classification models in which softmax is the final layer's activation function [4], [6], whereas NeuraLint supports both classification and regression models [2]. Finally, according to the analysis performed by Ghanbari *et al.*, shown in Table 1 (Appendix B), NeuraLint takes 2.87 seconds on average for fault localization, while UMLAUT takes substantially longer at 1302.61 seconds, which is primarily due to the sole reliance on static checks in NeuraLint [1].

4 Data & Configuration Fault Detection

DNNs are unable to learn effectively without the proper handling of data and correct configuration defined [8]. Errors related to data and configuration are a distinct class of faults, which often impact training time or reduce performance of a model [8]. These properties are generally defined before training begins and can be identified from the source code, as opposed to model bugs which may be detected from the trained model itself [1]. This section focuses on recent techniques that localize faults related to hyperparameter / training configuration bugs and dataset bugs, previously described in sections 2.1.2 through 2.1.3.

DeepFD and AutoTrainer are two recent dynamic techniques which are able to detect hyperparameter / training configuration bugs, but not dataset bugs [4], [10]. Both techniques are able to operate on fully connected, convolutional, and recurrent neural networks [4], [10]. However, the two techniques operate quite differently, with DeepFD framing fault localization as a learning problem [1], [4], while AutoTrainer monitors DNN state during the training process to detect issues and repair them [10].

DeepFD works by generating a diagnosis model, which is used to classify DNN programs into various fault categories that may be present, or classify as correct if no fault exists [4]. The diagnosis model is created by seeding faults into correct programs to produce training data, a technique which is considered a variant of mutation analysis [4]. This training data is used to train the diagnosis model and consists of information collected during the training process of seeded / unseeded models (e.g., loss, accuracy, gradients) with seeded faults used as labels [4]. The result is a diagnosis model which classifies DNN programs into categories of faults that are likely to be present, using features collected during training [4].

While AutoTrainer collects similar DNN state as DeepFD during the training process (e.g., loss, accuracy, gradients), AutoTrainer does not frame fault localization as a learning problem [10]. Instead, AutoTrainer functions by starting with an initial training configuration and utilizes state collected during training to identify any issues to be repaired [10]. Training will restart if a repair is applied, and the process will continue until the model is fixed or all repair attempts are unsuccessful [10]. Out of all techniques under review, AutoTrainer is one of two techniques which auto-repairs models (the other being NNRepair, covered in section 5).

Both DeepFD and AutoTrainer are able to identify faults related to inappropriate learning rate and wrong optimizer [4], [10]. In addition, DeepFD is able to identify faults related

to suboptimal loss function and insufficient iterations / epochs [4], while AutoTrainer can identify a suboptimal initializer, wrong batch size, and exploding gradients (resolved through gradient clipping) [10]. Although both techniques have the ability to identify hyperparameter / training configuration bugs in DNNs, DeepFD utilizes a learned diagnosis model to diagnose and localize bugs at the source code level, while AutoTrainer utilizes a set of rules to apply potential fixes to predefined issues [4], [10]. While AutoTrainer auto-resolves faults and may provide more actionable advice (e.g., set the learning rate to a specific range), DeepFD is able to generalize DNN faults without the use of predefined rules, and is likely easier to extend to additional model and fault types [4], [10].

DeepFD is quite efficient for fault localization of pre-trained models, as shown in Table 1 (Appendix D), taking 0.12 seconds or less on average to run the diagnosis model (training time of the diagnosis model is not taken into account) [4]. AutoTrainer is also efficient during its problem checking and repair steps, which take up about 1% of execution time [10]. However, it can be inferred that DeepFD is more efficient than AutoTrainer when taking training time into account, since AutoTrainer may require multiple retraining when applying fixes [4], [10]. However, once a fault is localized with DeepFD, the next logical step would be to retrain the model with a potential fix, so the overall time to resolve a bug with DeepFD may be greater, depending on the characteristics of the model being analyzed.

We now shift our attention to DeepCover, a dynamic fault localization approach which identifies dataset bugs in DNNs, but not hyperparameter / training configuration bugs [12]. DeepCover draws inspiration from techniques found in mutation analysis and spectrum-based fault localization, with the goal of explaining classifications from DNNs with image inputs [1], [12]. While DeepCover primarily supports convolutional neural networks, the algorithm treats the internals of DNNs as a black box, and so a variety of network structures are supported [12].

DeepCover functions by systematically mutating an image from a test set by masking a subset of pixels [12]. These mutated images are then classified by the DNN model, with the outputs compared to the output classification of the original image, with the goal of ranking pixels based on their importance for classification [12]. Using these pixel rankings, a minimum subset of pixels from the image is identified which is sufficient to classify the image, otherwise known as an *explanation* [12]. Thus, the output explanation may be interpreted as the region contributing most to the classification that is made [12]. This explanation could then be used to identify why certain inputs produce incorrect classifications, or whether a correct classification is made by “focusing” on the wrong regions [12]. In this way, DeepCover is able to localize dataset bugs, by identifying regions of input images contributing to incorrect and correct classifications [12]. It is important to note that DeepCover does not care whether an output classification is correct, and only cares to explain why a particular classification was made [12].

Finally, we cover UMLAUT (previously described in section 3), DeepDiagnosis, and DeepLocalize, which are DNN fault localization techniques that detect both hyperparameter / training configuration bugs and dataset bugs [3], [6], [7]. DeepDiagnosis and DeepLocalize are both dynamic techniques which are primarily compatible with fully connected and convolutional neural networks, with DeepDiagnosis possibly extended to recurrent neural networks in future iterations [6], [7]. DeepDiagnosis and DeepLocalize are similar in the way they function, as they both capture values during the feed-forward and back-propagation phases of training, such as weights, gradients, loss, and accuracy [6], [7]. Both DeepDiagnosis and DeepLocalize also utilize bug pattern rules to localize bugs, however, DeepDiagnosis takes it a step further by utilizing a decision tree to provide actionable fix suggestions, while having a more robust set of bug pattern rules [1], [6], [7]. A key limitation of

DeepLocalize is that it can only detect numerical errors during model training, while DeepDiagnosis is not subject to this limitation [1], [6], [7]. An additional limitation of DeepLocalize is mentioned by Cao *et al.*, stating that the localized numerical anomalies tend to not appear at the point where the true faults reside [4], further giving the edge to DeepDiagnosis, which is better able to localize faults and provide actionable fixes [6].

When considering hyperparameter / training configuration bugs, UMLAUT, DeepDiagnosis, and DeepLocalize are all able to localize faults related to an incorrect learning rate [3], [6], [7]. In addition, DeepDiagnosis is able to localize faults related to improper loss function, wrong learning rate, wrong optimizer, and incorrect batch size [6], while DeepLocalize is able to localize errors in the loss function [7].

Within the realm of dataset bugs, UMLAUT, DeepDiagnosis, and DeepLocalize are all able to localize faults related to an unnormalized dataset [3], [6], [7]. DeepDiagnosis, and DeepLocalize are able to localize improper input data (e.g., input contains `inf` or `NaN` values), while UMLAUT can localize validation set issues, which may arise if there is leakage between the training and validation data [3], [6], [7].

Concerning execution time of UMLAUT, DeepDiagnosis, and DeepLocalize, the analysis by Ghanbari *et al.* found that DeepDiagnosis is the most efficient of the three when a bug is successfully found, taking on average 11.05 seconds (overall average of 1510.71 seconds) [1]. DeepLocalize takes 57.29 seconds when a bug is successfully found (overall average of 1244.09 seconds), while UMLAUT is the least efficient, taking an average of 1302.61 seconds (no efficiency gain when a bug is successfully found) [1]. These execution time results are shown in Table 1 (Appendix B).

5 Model & Input Fault Detection

DNNs are unable to function correctly without a well-constructed model and valid inputs to layers [8]. Errors related to the model's structure or layer properties and problems with inputs form a distinct class of faults [8]. Unlike data and configuration faults which generally deal with properties defined before training, model and input faults can often be detected through analysis of the trained model itself, including its response to particular inputs [1]. This section focuses on recent techniques that localize faults related to model bugs and tensor / input bugs, previously described in sections 2.1.1 and 2.1.6.

NeuraLint and UMLAUT (both introduced in section 3) are two recent DNN fault localization techniques which are able to detect both model bugs and tensor / input bugs [2], [3]. In the realm of model bugs, both NeuraLint and UMLAUT have the ability to localize missing/redundant/wrong activation functions in the hidden and final layers, with a caveat being that UMLAUT only supports classification models with softmax as the final layer activation function (previously discussed in section 3) [2], [3]. While NeuraLint is able to localize issues related to an incorrectly placed dropout layer [2], UMLAUT is able to identify missing dropout layers, but only when overfitting is detected (in which case UMLAUT suggests adding dropout layers or reducing the complexity of the model) [3]. In addition to missing dropout layers, UMLAUT can identify dropout rates that are too high [3]. Meanwhile, NeuraLint can detect several model bugs that UMLAUT is incapable of, including: incorrect weight or bias initialization, incorrect inclusion of bias after a batch normalization layer, wrongly placed batch normalization layer, sub-optimal usage of down-sampling, excess usage of pooling operations after convolutions, and several other structural faults specific to convolutional neural networks [2].

Concerning tensor / input bugs, both NeuraLint and UMLAUT are able to localize issues with incorrect shape of image input data. NeuraLint also supports localization of several tensor / input bugs that UMLAUT is incapable of, including insufficiently sized feature space for spatial filtering or pooling, missing data elements after reshapes, and incorrect sizes of elements after reshapes. For additional analysis comparing NeuraLint and UMLAUT, including efficiency comparisons, section 3 may be referenced.

For the remainder of this section, we consider DNN fault localization techniques which are able to detect model bugs, but not tensor / input bugs. First we shall consider techniques which were previously described to also detect data and configuration faults, those being DeepFD, AutoTrainer, DeepDiagnosis, and DeepLocalize (all introduced in section 4). All four of these methods are able to localize faults related to incorrect activation functions in the final layer, with DeepDiagnosis and AutoTrainer also able to localize activation function faults in hidden layers [4], [6], [7], [10]. In addition, AutoTrainer is able to identify missing batch normalization layers [10]. Meanwhile, DeepDiagnosis is able to identify when the number of layers is too large or small, improper weight / bias initialization, as well as when the majority of nodes in a network are inactive (dead nodes) due to a large negative bias [6]. Section 4 may be referenced for additional analysis comparing DeepFD, AutoTrainer, DeepDiagnosis, and DeepLocalize, including efficiency comparisons.

The final set of recent DNN fault localization techniques includes those that are only able to detect model bugs out of all six categories of DNN faults. These techniques include deepmufl, NNRepair, and PAFL, all of which employ dynamic approaches for fault localization [1], [5], [11]. deepmufl is compatible with a variety of model types, including convolutional, recurrent, and fully connected neural networks for both classification and regression tasks [1]. NNRepair focuses on classifiers only, including convolutional and fully connected neural networks [5], while PAFL focuses solely on recurrent neural networks [11]. Although all three of these techniques focus on identifying model bugs, the methods used in each are quite different.

To achieve fault localization, deepmufl utilizes a variant of mutation analysis known as mutation-based fault localization [1]. Using a set of DNN mutators inspired by Humbatova *et al.* [9], deepmufl systematically mutates trained models to generate a set of mutated models [1]. For example, mutated models may include duplicated or deleted layers, or modifications to weights or biases [1]. This set of mutants is then executed against the test set to identify how performance differs compared to the original model [1]. Using this information, suspiciousness scores are calculated for each model element to identify locations of faults [1].

Comparing the efficiency of deepmufl to previously described techniques, NeuraLint, UMLAUT, DeepDiagnosis, and DeepLocalize are all more efficient in general, even when mutation selection is utilized in deepmufl [1]. However, the execution time of deepmufl can be substantially decreased without resulting in a large reduction in performance [1]. For example, when 50% of mutants are selected the average execution time decreases from 2192.40 seconds to 1714.63 seconds (taking training time into account), yet 92.45% of bugs can still be detected of those found with 100% of the mutants utilized [1]. In addition, deepmufl outperforms NeuraLint, UMLAUT, DeepDiagnosis, and DeepLocalize in terms of total model bugs localized [1]. The execution time results are shown in Table 1 (Appendix B), and the performance results of deepmufl are shown in Table 2 (Appendix B).

NNRepair not only localizes faults, but also executes repairs on the model, similar to AutoTrainer [5], [10]. During its fault localization step, NNRepair analyzes activation patterns of intermediate layers and the output of the network from the final layer to identify neurons and

their incoming edges that may be faulty [5]. This information is used to generate a set of repair constraints, which are then solved to determine weight updates that are applied to the model [5].

`PAFL`, like most of the techniques under review, localizes faults but does not initiate repairs [11]. `PAFL` functions by transforming a recurrent neural network into a probabilistic finite automaton, which is then utilized to localize sequences of state transitions as faults of the network [11]. In addition, a suspiciousness score is calculated (using spectrum-based fault localization) which identifies data samples contributing to faulty states [11]. Comparing `PAFL` to `DeepCover` (described in section 4), the identification of data samples contributing to faulty states in `PAFL` is somewhat similar to the identification of regions in input images contributing to classifications in `DeepCover` [11], [12]. However, `PAFL` does not mutate inputs as `DeepCover` does to achieve explanations [11], [12]. In addition, explanations created by `DeepCover` do not automatically convey that a fault is present, as these explanations are just as valid for correct classifications [12]. Meanwhile, the data samples `PAFL` identifies indicate that a fault occurs when they are used as input to the model [11].

Comparing the types of model bugs that are localized by `deepmufl`, `NNRepair`, and `PAFL`, both `deepmufl` and `NNRepair` are able to identify incorrect weight initializations, with `NNRepair` actually repairing weights in models [1], [5]. `PAFL` is unique across all reviewed fault localization techniques, since it localizes sequences of state transitions as faults of recurrent neural networks, while also identifying data samples contributing to faulty states [11]. Finally, `deepmufl` localizes several faults that `NNRepair` and `PAFL` cannot localize, including improper activation functions, incorrect bias values, wrong filter/kernel/stride size, sub-optimal number of neurons, and missing/redundant/wrong layer [1].

6 Experiment

This experiment aims to replicate key findings from the paper by Ghanbari *et al.* on mutation-based fault localization applied to DNNs [1]. In particular, the impact of mutation selection on the effectiveness and execution time of `deepmufl` was tested and compared to the original results from their paper, which are presented in Table 2 (Appendix B). Finally, a discussion of the results including potential areas of future work are discussed in section 6.4.

6.1 Methodology

The source code of `deepmufl` was downloaded from the repository [13] referenced in the original paper by Ghanbari *et al.* [1]. Due to issues with satisfying requirements for the Conda virtual environment in Windows and macOS, VirtualBox 7.0.14 was used to create a virtual machine running the Linux distribution Ubuntu 22.04.4 LTS. The machine used to run VirtualBox was running Windows 10 with an Intel Core i7-8700K processor overclocked to 4.77 GHz, 16 GB of RAM, an NVIDIA GeForce RTX 2080 graphics card, and a 500 GB M.2 SSD. However, in order to suppress warnings in VirtualBox, only 11,428 MB of base memory, 6 CPUs, and 16 MB of video memory was allocated to the virtual machine.

Although Ubuntu 22 was also originally used by Ghanbari *et al.* [13], several requirements specified in the original repository did not automatically resolve to a valid Conda virtual environment. Using the output Conda error messages, dependencies were updated one by one to the nearest compatible version, until the virtual environment was successfully created. Additionally, once the environment resolved, the `deepmufl` source code did not automatically work on all inputs. After some investigation, it was determined that several bugs were present in the mutation generator code, including:

- Hardcoded logic in the delete layer mutation logic. This issue was unable to be resolved, so the logic was removed from the pool of mutators.
- Issues with the duplicate layer mutation logic getting applied to certain layers. This issue was unable to be resolved, so the logic was removed from the pool of mutators.
- Incorrect usage of an internal function which adds mutated models to a TAR file, resolved by correctly calling the function with the path of the model file to save.
- Issues with adding a large number of mutated models to the TAR file in quick succession, due to immediate deletion of model files after each addition. This was resolved by adding a delay of 0.1 seconds after adding each model to the TAR file. This issue is likely to have occurred due to the use of a virtual environment, which may have impacted read/write speeds. Since this delay was artificially added, the total time delay has been removed from the experiment results in Appendix E.

Since several changes were applied, the modified `deepmufl` source code is included in a personal GitHub repository [14], along with the dataset of models utilized and results.

Source code of the models used in the experiment were obtained from the original repository [13]. The source code of each model was modified so that the test set inputs, test set outputs, and trained model were stored as files, with each replicated three times (these files are necessary inputs to `deepmufl`). In addition, for any models that did not incorporate a training / test split, the code was modified to include 20% of the data as the test set, if possible (if not possible, the entire dataset of inputs and outputs was saved). A random state of 42 was also added to all train / test splits. The inputs, outputs, trained models, and Jupyter notebook used for generating these files are included in the personal GitHub repository [14].

Each trained model (including replicated models) and their associated inputs and outputs were used as inputs to `deepmufl`, with each set of inputs run four times (for each of 25%, 50%, 75%, and 100% rates set for mutation selection). Depending on whether each model was a classification or regression model, the `class` or `reg` parameter was set accordingly when running `deepmufl`. Finally, a value of 0.001 was used as the delta value parameter for comparing floating point values, which is the default setting of `deepmufl`. The output files containing the fault localization results were analyzed via a Python script to extract results and determine whether fault localization was successful or unsuccessful. A fault was considered to be localized if the suspiciousness core of the layer actually containing the fault was the maximum score across all layers, and no ties were present across layers with this maximum score. The output files and script are available in the personal GitHub repository [14].

6.2 Dataset of DNN Bugs

The output of `deepmufl` consists of average and max values at each layer using the Metallaxis SBI type 1 and type 2 formulas, Metallaxis Ochiai type 1 and type 2 formulas, and MUSE formula. Since the `deepmufl` outputs only specify suspiciousness scores at each layer and not specific faults contributing to those scores, it was particularly difficult to determine whether a fault was actually localized, especially when multiple faults existed in a network at the same layer. Thus, the decision was made to only focus on models with a single fault, which was determined by inspecting the top answers from each model's associated Stack Overflow post. In addition, only faults related to an incorrect activation function were considered. In total, 6 classification models and 6 regression models were selected from the original dataset of model bugs from Ghanbari *et al.* [13], with all except one of the regression models containing the fault in the final layer. 5 of these models incorrectly used softmax, 4 incorrectly used sigmoid, 2 incorrectly used ReLU, and 1 incorrectly used a linear activation function in a hidden layer.

6.3 Results

Overall results from this experiment are displayed in Table 1 (Appendix E), which displays the number of faults that were detected using each of Metallaxis SBI type 1 and type 2 formulas, Metallaxis Ochiai type 1 and type 2 formulas, and the MUSE formula, in addition to execution times. Since both average and max scores were reported at each layer for the Metallaxis calculations, fault localization using average and max scores are reported separately. The original paper by Ghanbari *et al.* [1] does not make a distinction between average and max scores, so these additional rows are not present in Table 2 (Appendix E), which was generated using the original data collected from Ghanbari *et al.* [13] by only including results from the 12 models used in this experiment. Section 6.4 compares the results from the average and max calculations in Table 1 (Appendix E) from this experiment, and also compares these results to Table 2 (Appendix E) containing the overall results from the original paper.

In addition to overall results, Table 3 (Appendix E) only includes experiment results from the 6 classification models in the experiment, with Table 4 (Appendix E) included for comparison to the 6 classification model results from the original paper. Similarly, Table 5 (Appendix E) only includes experiment results from the 6 regression models in the experiment, with Table 6 (Appendix E) included for comparison to the 6 regression model results from the original paper. Separating the classification and regression results in this way allows for a comparison of fault localization effectiveness and efficiency between these two model types, an aspect which is not covered in the paper by Ghanbari *et al.* [1]. We cover this aspect in the discussion.

Finally, it is important to note that the experiment included 3 replicas for each model, so in actuality there are 36 total results instead of 12. To account for this, Table 1 (Appendix E), Table 3 (Appendix E), and Table 5 (Appendix E) each divide the number of detected faults by 3, so that direct comparisons may be made to Table 2 (Appendix E), Table 4 (Appendix E), and Table 6 (Appendix E), respectively. Thus, the maximum score from the first two tables is 12, and the maximum score is 6 for the final four tables.

6.4 Discussion

Looking at the results from this experiment in Table 1 (Appendix E), Table 3 (Appendix E), and Table 5 (Appendix E), fault localization effectiveness is generally higher when using average scores compared to max scores for the Metallaxis SBI and Ochiai type1 and type 2 formulas. This observation is consistent across both classification and regression models. However, the original paper by Ghanbari *et al.* defines the SBI and Ochiai formulas with the max operation, so whether using average is a valid modification to these formulas for fault localization is up for debate, and potentially an area of future research.

Concerning performance differences of `deepmufl` as the number of selected mutants changes, a key metric from the original paper by Ghanbari *et al.* was that 92.45% of faults detected with 100% of mutants selected could still be detected when 50% of mutants were selected using the MUSE configuration (see Table 2, Appendix B for this result). From Table 2 (Appendix E) which shows the overall results from the original paper for the 12 models used in the experiment, the 92.45% metric increases to 100%, due to the nature of the 12 models that were selected. Comparing this result to the results in Table 1 (Appendix E), the experimental

result only achieves 90.43% for the same metric. This performance is only about 2% less than the result from the original paper, but it does underperform compared to the expected result.

Concerning the effect of mutation selection on execution time, we observe a peculiar result. While the classification models take longer to execute across all proportions of selected mutants in the experimental results (Table 3, Appendix E) compared to the results from the original paper (Table 4, Appendix E), the regression models take less time to execute in the experimental results (Table 5, Appendix E) compared to the original paper results (Table 6, Appendix E). This inconsistency is unexpected, but possibly due to hardware differences which allow greater efficiency on the regression models in the experimental results. We also observe that execution time takes longer for classification models when 25% of mutants are selected when compared to 75% of mutants selected (Table 3, Appendix E). However, when taking a closer look at the raw recorded results, this is caused by a single classification model which takes 18851.28 seconds to run [14], which is possibly caused by system issues. Despite these surprises, the overall trend in the experimental results is consistent with the results from the original paper, that being that execution time decreases as selected mutants decrease, and that execution time can be halved when utilizing 50% of mutants compared to 100% of mutants.

Finally, an interesting result is revealed from tables 3 through 6 in Appendix E. The MUSE formula performs best for classification models, which is supported by both the experimental results in Table 3 (Appendix E) and the original paper results in Table 4 (Appendix E). In comparison, for regression models MUSE performs poorly, while Metallaxis Ochiai type 2 performs best, which is supported by both the experimental results in Table 5 (Appendix E) and the original paper results in Table 6 (Appendix E). This difference is not explored in the original paper by Ghanbari *et al.*, which suggests that MUSE achieves the highest average performance across all model types in their dataset [1]. A potential area of future research is to discover which of these formulas apply best to certain model structures, in order to intelligently choose the best formula for fault localization.

7 Conclusion

Fault localization techniques generally fall into three high level categories: those which localize (1) faults during system interactions, (2) data or configuration faults, and (3) model structure or input faults. Recent DNN fault localization techniques utilize a variety of static and/or dynamic approaches to localize DNN bugs. Those that employ static checks are generally efficient. However, relying on static checks may miss certain attributes of DNNs that can only be observed through their behavior. Throughout this paper we have gained an understanding of the strengths and weaknesses of each fault localization technique, which is crucial for understanding when to apply these approaches to resolve complex bugs in DNNs.

What I Learned: I gained a deep understanding of recent fault localization approaches, including the faults each technique localizes, how each technique operates, efficiency comparisons, and strengths / weaknesses. I am more confident in my ability to apply these approaches to debug DNNs. Finally, I learned about the `deepmufl` source code, and I am excited to improve the logic so that fault localization is more effective. The most surprising finding was that the MUSE formula does not universally perform better across all model types, which was not conveyed by the original paper on `deepmufl`. I also learned the difficulty of replicating results, as evidenced by the many changes made to the source code.

References

[1] A. Ghanbari, D. -G. Thomas, M. A. Arshad and H. Rajan, "Mutation-based Fault Localization of Deep Neural Networks," 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg, 2023, pp. 1301-1313. https://doi.org/10.1109/ASE56229.2023.00171

[2] 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. https://doi.org/10.1145/3470006

[3] E. Schoop, F. Huang, and B. Hartmann, “Umlaut: Debugging deep learning programs using program structure and model behavior,” in CHI, 2021, pp. 1–16. https://doi.org/10.1145/3411764.3445538

[4] 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. https://doi.org/10.1145/3510003.3510099

[5] 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. https://doi.org/10.1007/978-3-030-81685-8_1

[6] 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. https://doi.org/10.1145/3510003.3510071

[7] M. Wardat, W. Le, and H. Rajan, “Deeplocalize: fault localization for deep neural networks,” in ICSE, 2021, pp. 251–262. https://doi.org/10.1109/ICSE43902.2021.00034

[8] 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. https://doi.org/10.1145/3377811.3380395

[9] N. Humbatova, G. Jahangirova, and P. Tonella, “Deepcrime: mutation testing of deep learning systems based on real faults,” in ISSTA, 2021, pp. 67–78. https://doi.org/10.1145/3460319.3464825

[10] 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. https://doi.org/10.1109/ICSE43902.2021.00043

[11] 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. https://doi.org/10.1016/j.infsof.2022.107117

[12] Y. Sun, H. Chockler, X. Huang, and D. Kroening, “Explaining image classifiers using statistical fault localization,” in ECCV, 2020, pp. 391–406. https://doi.org/10.1007/978-3-030-58604-1_24

[13] 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.

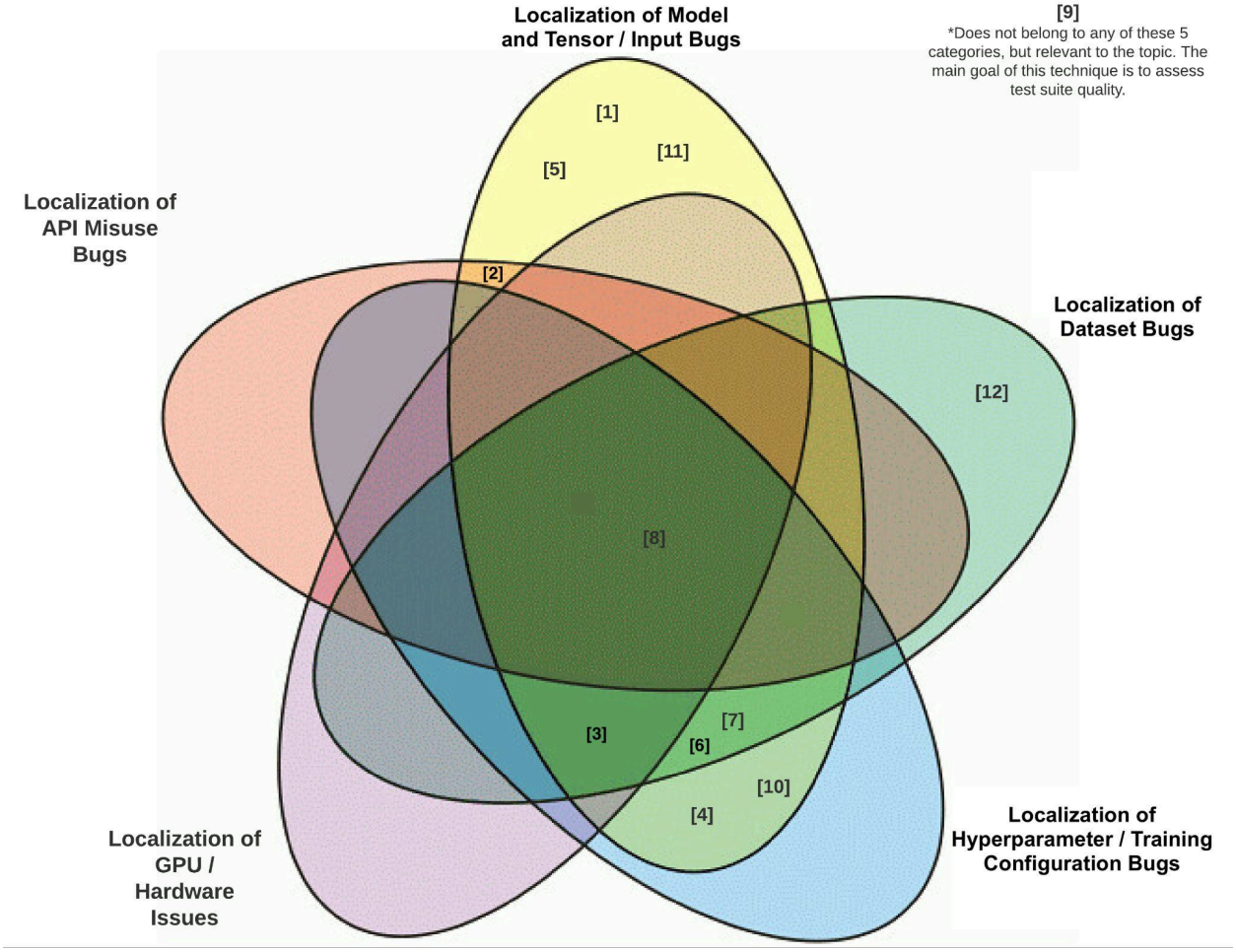
[14] R. Arora, “deepmufl-experiment” https://github.com/rarora9/deepmufl-experiment, 2024.

Appendix A

Venn Diagram of References

What has changed:

- *"Localization of Model Bugs" category changed to "Localization of Model and Tensor / Input Bugs".*
- *"Localization of Training Data Bugs" category changed to "Localization of Dataset Bugs".*
- *"Localization of Hyperparameter Bugs" category changed to "Localization of Hyperparameter / Training Configuration Bugs".*
- *Moved the following references: [2], [3], [6].*



Appendix B

Supporting Materials from Ghanbari *et al.* [1]

Table 1

	deepmufl (25% mutants)*	deepmufl (50% mutants)*	deepmufl (75% mutants)*	deepmufl (100% mutants)*	Neuralint	DeepLocalize	DeepDiagnosis	UMLAUT
Avg. time (s)	1492.48	1714.63	1958.35	2192.40	2.87	1244.09	1510.71	1302.61
Avg. time (success) (s)	N/A	N/A	N/A	N/A	N/A	57.29	11.05	N/A

* `deepmufl` average times include training time of the model.

Table 2

	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

Appendix C

Supporting Materials from Nikanjam *et al.* [2]

Table 1

Execution Time of *NeuraLint* for Five Real DL Programs with Different Sizes (Times are in Seconds)

No.	Number of layers	Running time of Graph extraction	Running time of Graph checking
1	6	0.003	1.757
2	8	0.003	1.787
3	12	0.002	1.836
4	13	0.003	1.906
5	38	0.004	3.111

Appendix D

Supporting Materials from Cao *et al.* [4]

Table 1

Effectiveness of Diagnosis Models of DeepFD on Labeled DNN Set. KNN, DT and RF stand for the underlying algorithms (K-Nearest Neighbors, Decision Tree and Random Forest) of diagnosis models.

Dataset	Statistics								Accuracy and Average Runtime of Diagnosis Models					
	Origin	Mutant	# Types of Faults					Time	KNN		DT		RF	
			1	2	3	4	5		Acc (%)	Time (s)	Acc (%)	Time (s)	Acc(%)	Time (s)
MNIST	78	1768	1027	302	295	134	10	0.60	69.29	0.10	63.15	0.01	81.58	0.12
CIFAR-10	35	786	651	102	33	0	0	1.56	52.63	0.05	53.94	0.02	64.47	0.12
Circle	36	936	580	174	133	44	5	8.87	45.20	0.23	55.50	0.03	61.36	0.22
Blob	39	685	335	158	137	50	5	0.13	83.95	0.06	79.01	0.01	87.65	0.10
Reuters	32	175	138	22	3	12	0	36.30	79.69	0.05	79.69	0.01	81.25	0.09
IMDB	13	110	76	25	9	0	0	53.72	**93.30**	0.04	**88.80**	0.03	**93.30**	0.09
Total	233	4,460	2,807	783	610	240	20	Average	**70.68**	0.09	**79.90**	0.02	**78.27**	0.12

Appendix E

Experiment Results

Table 1

Table 1: Experimental results for the impact of mutation selection on the effectiveness and average execution time of `deepmufl` for classification and regression models with incorrect activation function faults in a single layer. The number of detected faults are divided by three in order to make direct comparisons to Table 2, since each model was trained three times.

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1 + Avg	3.33	4.33	4.67	4.67
Metallaxis Ochiai + Type 1 + Avg	5	5.67	6	5.67
Metallaxis SBI + Type 2 + Avg	3.33	4.67	6.33	6.33
Metallaxis Ochiai + Type 2 + Avg	6	7.33	8	8
Metallaxis SBI + Type 1 + Max	2	2	2.33	2.33
Metallaxis Ochiai + Type 1 + Max	3.33	3	3.33	3.67
Metallaxis SBI + Type 2 + Max	0.67	1.33	1.33	1.33
Metallaxis Ochiai + Type 2 + Max	4.67	5.33	5.33	2.67
MUSE	5.67	6.33	6.67	7
Time (s)	1066.16	736.30	905.76	1545.87

Table 2

Table 2: The impact of mutation selection on the effectiveness and average execution time of `deepmufl` for classification and regression models with incorrect activation function faults in a single layer. Based on the results from Ghanbari *et al.* [1], [13].

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1	4	3	3	3
Metallaxis Ochiai + Type 1	3	3	3	3
Metallaxis SBI + Type 2	3	3	3	3
Metallaxis Ochiai + Type 2	5	7	7	7
MUSE	6	7	7	7
Time (s)	265.49	525.83	768.00	1010.39

Table 3

Table 3: Experimental results for the impact of mutation selection on the effectiveness and average execution time of `deepmufl` for classification models with incorrect activation function faults in the final layer. The number of detected faults are divided by three in order to make direct comparisons to Table 4, since each model was trained three times.

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1 + Avg	3.33	4	4	4
Metallaxis Ochiai + Type 1 + Avg	5	5.33	5.33	5
Metallaxis SBI + Type 2 + Avg	1	1	1	1
Metallaxis Ochiai + Type 2 + Avg	2	2	2	2
Metallaxis SBI + Type 1 + Max	2	2	2	2
Metallaxis Ochiai + Type 1 + Max	3.33	3	3	3.33
Metallaxis SBI + Type 2 + Max	0	0	0	0
Metallaxis Ochiai + Type 2 + Max	1.33	1.33	1.33	0.67
MUSE	5.67	6	6	6
Time (s)	2107.70	1432.39	1755.98	3020.03

Table 4

Table 4: The impact of mutation selection on the effectiveness and average execution time of `deepmufl` for classification models with incorrect activation function faults in the final layer. Based on the results from Ghanbari *et al.* [1], [13].

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1	3	2	2	2
Metallaxis Ochiai + Type 1	2	2	2	2
Metallaxis SBI + Type 2	1	0	0	0
Metallaxis Ochiai + Type 2	0	1	1	1
MUSE	5	6	6	6
Time (s)	493.47	978.55	1441.05	1878.61

Table 5

Table 5: Experimental results for the impact of mutation selection on the effectiveness and average execution time of `deepmufl` for regression models with incorrect activation function faults in a single layer. The number of detected faults are divided by three in order to make direct comparisons to Table 6, since each model was trained three times.

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1 + Avg	0	0.33	0.67	0.67
Metallaxis Ochiai + Type 1 + Avg	0	0.33	0.67	0.67
Metallaxis SBI + Type 2 + Avg	2.33	3.67	5.33	5.33
Metallaxis Ochiai + Type 2 + Avg	4	5.33	6	6
Metallaxis SBI + Type 1 + Max	0	0	0.33	0.33
Metallaxis Ochiai + Type 1 + Max	0	0	0.33	0.33
Metallaxis SBI + Type 2 + Max	0.67	1.33	1.33	1.33
Metallaxis Ochiai + Type 2 + Max	3.33	4	4	2
MUSE	0	0.33	0.67	1
Time (s)	24.61	40.21	55.55	71.72

Table 6

Table 6: The impact of mutation selection on the effectiveness and average execution time of `deepmufl` for regression models with incorrect activation function faults in a single layer. Based on the results from Ghanbari *et al.* [1], [13].

	Selected mutants			
	25%	**50%**	**75%**	**100%**
Metallaxis SBI + Type 1	1	1	1	1
Metallaxis Ochiai + Type 1	1	1	1	1
Metallaxis SBI + Type 2	2	3	3	3
Metallaxis Ochiai + Type 2	5	6	6	6
MUSE	1	1	1	1
Time (s)	37.51	73.12	94.95	142.17