Leveraging Data Characteristics for Bug Localization in Deep Learning Programs

RUCHIRA MANKE, Department of Computer Science, Tulane University, USA
MOHAMMAD WARDAT, Department of Computer Science and Engineering, Oakland University, USA
FOUTSE KHOMH, SWAT Lab., Polytechnique Montréal, CA
HRIDESH RAJAN, School of Science and Engineering, Tulane University, USA

Deep Learning (DL) is a class of machine learning algorithms that are used in a wide variety of applications. Like any software system, DL programs can have bugs. To support bug localization in DL programs, several tools have been proposed in the past. As most of the bugs that occur due to improper model structure known as structural bugs lead to inadequate performance during training, it is challenging for developers to identify the root cause and address these bugs. To support bug detection and localization in DL programs, in this paper, we propose Theia, which detects and localizes structural bugs in DL programs. Unlike the previous works, Theia considers the training dataset characteristics to automatically detect bugs in DL programs developed using two deep learning libraries, *Keras* and *PyTorch*. Since training the DL models is a time-consuming process, Theia detects these bugs at the beginning of the training process and alerts the developer with informative messages containing the bug's location and actionable fixes which will help them to improve the structure of the model. We evaluated Theia on a benchmark of 40 real-world buggy DL programs obtained from *Stack Overflow*. Our results show that Theia successfully localizes 57/75 structural bugs in 40 buggy programs, whereas NeuraLint, a state-of-the-art approach capable of localizing structural bugs before training localizes 17/75 bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: deep learning bugs, bug localization, debugging, program analysis

1 INTRODUCTION

Deep learning (DL) based software has recently gained popularity and is being used in various fields, including chatbots [79], virtual assistants [59], and financial institutions [80]. Their popularity has drawn the interest of the software engineering community to understand their development process. As bugs are inherent to the software development process, several studies have been conducted in the past to understand the characteristics of DL bugs, their root causes, and repair solutions [58, 61, 62, 90]. To support the development of DL programs, several DL libraries and frameworks, such as Tensorflow [50], Keras [51], and Pytorch [78] are available which provide various APIs for building, training, and evaluating these programs. As DL programs are based on tensor operations, which are multi-dimensional arrays that generalize matrices to higher dimensions, various operations on tensors, such as matrix multiplication, addition, activation functions, and convolutions are performed to build

Authors' addresses: Ruchira Manke, rmanke@tulane.edu, Department of Computer Science, Tulane University, 303 Stanley Thomas Hall, New Orleans, Louisiana, 70118, USA; Mohammad Wardat, wardat@oakland.edu, Department of Computer Science and Engineering, Oakland University, 115 Library Drive, Rochester, MI, 48309, USA; Foutse Khomh, foutse.khomh@polymtl.ca, SWAT Lab., Polytechnique Montréal, P.O. Box 6079, Station Centre- Ville, Montréal, H3C 3A7, CA; Hridesh Rajan, hrajan@tulane.edu, School of Science and Engineering, Tulane University, 201 Lindy Boggs Center, New Orleans, Louisiana, 70118, USA.

ACM 1557-7392/2024/12-ART
https://doi.org/10.1145/3708473

and train these models. These libraries validate the correctness of the computations and use assertions to detect crash bugs. However, the dependency of DL programs on data makes it challenging to impose assertions for silent bugs, that occur due to hidden logic errors and commonly lead to incorrect model behavior, inaccurate predictions, or degraded performance. Although, these libraries provide *'callbacks'* to monitor and customize various stages of training loops (*e.g.,* at the start or end of an epoch, at the start or end of the batch, *etc.*), these callback methods (*e.g.,* EarlyStopping(), TerminateOnNaN()) do not indicate which layer or hyper-parameter caused the issue. As a result, DL libraries lack comprehensive debugging mechanisms for locating silent bugs. Researchers in the past [58, 61] have found that silent bugs are more prevalent (> 60%) than crash bugs in DL programs. In the software engineering community, these bugs are referred to as *structural bugs* [71]. In this work, we focus on structural bugs, which primarily arise from misconfigured hyper-parameters in the model.

To assist developers in identifying the structural bugs in DL programs, several techniques such as UMLAUT [81], DeepLocalize [88], DeepDiagnosis [87], TheDeepChecker [42], DeepFD [46] for detecting and localizing these bugs have been proposed in the past. These techniques observe the abnormal behavior during the training of the model and identify structural bugs based on certain symptoms. Due to one-to-many mapping between the abnormal behavior observed during training and its root causes [87], these techniques are unable to provide sufficient insight into the underlying cause of the issue, thereby requiring several rounds to fix the bugs. As training a DL model is expensive, identifying model inefficiencies during training wastes computational resources. To overcome this problem, Nikanjam *et al.* [76] proposed a static approach, NeuraLint, that examines the DL model for structural errors and design inefficiencies and can detect bugs in DL programs that are not covered by previous dynamic approaches [46, 81, 87, 88].

The current approaches for detecting and localizing bugs in DL programs are either specifically designed for classification tasks [81, 89] or focused on identifying structural bugs that are common across different DL architectures (*i.e.,* Fully-Connected Neural Networks (FCNN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN)) [42, 46, 87, 88]. NeuraLint [76] covers CNN architecture-specific structural bugs. However, it relies on the parsed source code of the DL model and does not consider the training data to localize the bugs, resulting in false alarms. As the DL models are data-driven, information acquired using only the parsed source code is insufficient for effective bug localization. Our insight is that, as the DL models are data-driven, combining the training data characteristics with the model's source code provides a more comprehensive analysis, improving the bug localization accuracy.

1.1 Motivation

In practice, new developers usually use familiar solutions when designing the DL program without fully understanding the effect of those solutions and various factors that need to be considered for their application. For instance, utilizing the architecture of the model designed for multi-class classification for binary classification or a simple model designed for gray-scale image datasets for more complex color images leads to inadequate performance during training. Due to the numerous hyper-parameters in DL models, debugging these programs is challenging [58, 62]. For instance, Fig. 1 presents a query posted on *Stack Overflow* [16], wherein the developer has implemented a CNN program using *Keras*. The program demonstrated erratic behavior during training and testing. The developer in the post mentions the various CNN architectures that were attempted but did not achieve the desired results. In response, another *Stack Overflow* user pointed out the following issues in the CNN program. First, the dataset used to train the model is very small, and the user recommends increasing the dataset size by adding high-quality data. Secondly, as the model is designed for multi-class classification, the user also suggests improving the network design by adding more convolution layers and explaining the impact of shallow and overly deep networks on performance. As a side note, the user also suggests lowering the learning rate. The

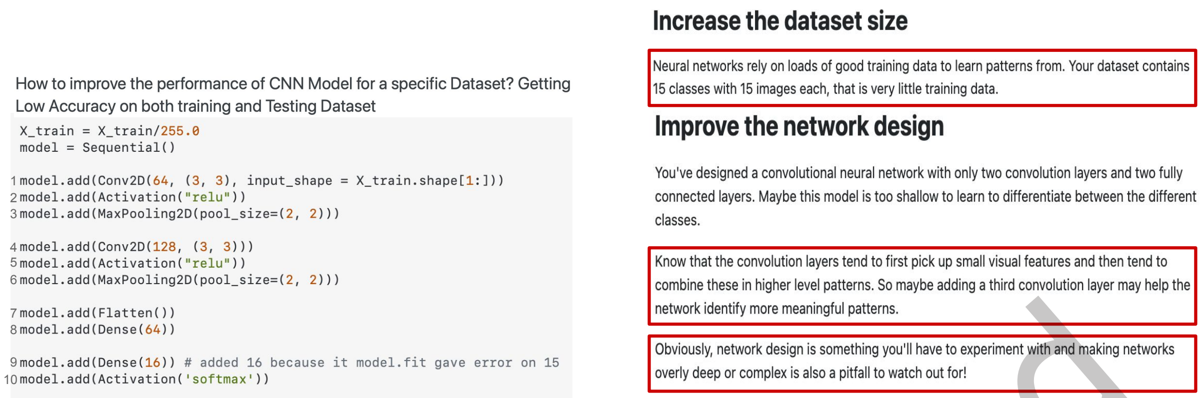


Fig. 1. An example from *Stack Overflow* [16] with structural bug.

developer could not design a CNN program with good performance and locate the cause of these bugs because structural bugs usually affect the program's performance rather than causing the program to crash.

For the CNN program in Fig. 1, current state-of-the-art techniques [42, 46, 81, 87–89] were not able to identify structural flaws, as mentioned in the *Stack Overflow* post. These techniques primarily focus on identifying bugs using different parameters such as weights, gradients, loss, and accuracy within the designed model, assuming that the model's depth and width are appropriately defined. Therefore, these techniques do not identify structural flaws due to suboptimal model depth or width, which can significantly affect performance, as shown in Fig. 1. While NeuraLint identifies structural errors before training, as it relies on the model's parsed source code and does not capture the characteristics of the training dataset in the meta-model, it cannot determine whether the model is too shallow or narrow for the training dataset. For the example in Fig. 1, existing approaches cannot identify the structural bug due to the insufficient number of layers in the model. Choosing an appropriate number of layers or neurons during model design is challenging. In practice, it is usually done by manually fine-tuning the model or using automated tools like Auto-Keras [63]. However, fine-tuning is expensive [60]; on high-performance machines, Auto-Keras usually requires 8-12 hours to search for models with reasonable accuracy (90% or higher) [63]. Developers often need to pay more attention to the fundamental design principles, which leads to incorrect model behavior during training and requires significant debugging time and effort. Therefore, some lightweight automated debugging tools are needed to verify the designed model structure aligns with the training dataset and task before initiating an expensive training process.

1.2 Contributions

In this paper, we propose a technique, named Theia, which leverages the characteristics of the training dataset along with the model's parsed source code for localizing the structural bugs in DL programs, *i.e.,* bugs related to the activation function, layer properties, model properties, loss function, preprocessing of data, and bugs due to missing/redundant/wrong layers and provide suggestions to fix the bug. These bugs lead to performance issues, *i.e.,* low/stuck accuracy during training. Therefore, the scope of Theia is to localize structural bugs in DL programs. Theia supports two types of DL architecture, *i.e.,* Fully-Connected Neural Networks (FCNNs) and Convolutional Neural Networks (CNNs) designed for regression, as well as classification tasks.

To design Theia, first, a general representation of the DL program, a *meta-model*, that is independent of any DL libraries or frameworks is constructed. A meta-model captures the characteristics of the dataset, *e.g.,* dimension, type of training data, and properties of the DL model, *e.g.,* the number of convolution layers and learning rate. Theia utilizes the meta-model and performs context-sensitive analysis, namely call-strings analysis [73] and parameter-sensitive analysis [84] using the verification rules to detect the structural bugs in DL programs developed using two popular deep learning libraries, *Keras* and *PyTorch*. Theia detects these bugs at the beginning of the training process and alerts the developer with informative messages that include the bug's location and fix recommendations to improve the structure of the DL model.

We evaluated Theia on 40 real-world buggy DL programs obtained from *Stack Overflow* designed for regression and classification tasks. Theia successfully finds 57/75 bugs and is more effective than NeuraLint [76], which detects 17/75 bugs.

In summary, this paper makes the following contributions:

- We investigated the mapping between the characteristics of the dataset and the structure of the model.
- We provide verification rules to detect the occurrence of structural bugs.
- We designed and implemented Theia, for two popular DL libraries, *Keras* and *PyTorch*, for automatically detecting structural bugs at the beginning of the training process.
- We evaluated Theia on 40 buggy DL programs and compared with NeuraLint [76]. We found that Theia is more effective and efficient compared to NeuraLint which can be used by developers to detect structural bugs in DL programs.

The rest of the paper is organized as follows. §2 describes the background. §3 describes the deep learning program structural bugs. §4 describes the verification rules, explains how they are used to detect structural bugs in our approach, and presents an algorithm for identifying these bugs. §5 describes the evaluation of our approach compared with prior work. §6 discusses the threats to validity. §7 discusses related work, §8 concludes and discusses future work, and §9 provides details of the replication package.

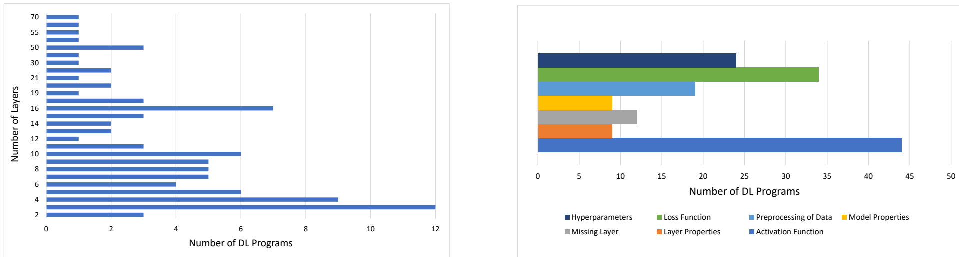
2 BACKGROUND

2.1 Deep Learning Programs

Deep learning has recently been widely used in different domains to automatically learn complex patterns from data [46]. Deep learning architectures (*i.e.,* FCNN, CNN) comprise many layers with each layer serving a distinct function. These layers are fundamental building blocks that transform input data into meaningful output. For example, the FCNN program comprises an input layer followed by a series of fully connected layers that learn features from input, and finally an output layer that is trained to predict the output. CNN program has a more complex structure comprising convolution and pooling layers followed by fully connected layers. Convolution layers extract the features from the input and produce feature maps, pooling layers help in reducing the size of feature maps, and fully connected layers help the model learn class-wise features. Using features extracted from previous layers, the output layer is trained to predict the final output. The DL program has two types of parameters: (1) the model parameters that are learned during training; and (2) the hyper-parameters whose value can be configured before training, *e.g.,* number of neurons, filters in the convolution layer, kernel size, strides. Each layer has a different number of hyper-parameters that help the model to learn and are provided by the developer while designing the DL programs.

2.2 Deep Learning Library

Keras and *PyTorch* are the popular deep learning libraries that provide APIs for implementing different stages of a DL program, namely, data preparation, modeling, and training [44]. The APIs are written in the form of classes, *e.g.,* `Conv2D`, `MaxPool2d`, `Dense`, *etc.,* comprising many methods with parameters. Some of these parameters are used in structuring the model, while others serve as *hyper-parameter*s for the DL program which helps in learning. The hyper-parameters are initialized by the developer while using the API and must be configured considering the task for which the DL program is designed as well as the characteristics of the training dataset. Failing



(a) Distribution of different types of DL programs with varying depths

(b) Distribution of bugs in different categories

Fig. 2. Details of DL programs used for mapping.

to consider them while designing the DL programs results in the incorrect configuration of hyper-parameters which may not necessarily cause a program to crash, however, results in a program with performance issues, *e.g.*, incorrect output or low/stuck classification accuracy.

3 DEEP LEARNING PROGRAM BUGS

In this section, we first describe the process utilized to identify the structural bugs that lead to performance issues. Then, we discuss the mapping between the characteristics of the dataset and the structural bug. Followed by verification rule creation methodology.

3.1 Structural Bugs Investigation

Researchers in the past have studied deep learning program bugs, and their characteristics and also provided a taxonomy of faults for these programs. Zhang *et al.* [90] studied the root cause of bugs and their symptoms in TensorFlow programs. This research was extended by Islam *et al.* [61] and they studied the types of bugs, their root causes, and their impacts using five popular DL libraries. Humbatova *et al.* [58] further refined the bug investigation and provided a taxonomy of real faults in deep learning systems. The taxonomy was derived using 375 buggy program posts obtained from *Stack Overflow* and *GitHub* designed using three popular DL libraries: Tensorflow, Keras, and PyTorch. Moreover, the taxonomy was further enhanced by conducting interviews with 20 researchers and validated by involving an additional 21 developers. The taxonomy is broadly classified into five categories: *Model, GPU usage, API, Tensors & Inputs, and Training*. The structural bugs (bugs due to suboptimal model structure), might appear in any of the five categories and can cause crashes or poor/unexpected accuracy when the DL model is trained. The bugs in some of these categories, *i.e.*, '`GPU Usage`', '`API`', and '`Tensors & Inputs`' cause the DL program to crash while bugs in categories '`Model`' and '`Training`' typically result in low/stuck accuracy during training. While non-crashing bugs do not raise an exception, they negatively impact training and lead to poor generalization, crash bugs raise an exception during compilation/execution, *e.g.*, tensor shape mismatch, or deprecated API. We relied on the taxonomy provided by [58] and focused on non-crashing bugs in this paper.

3.2 Mapping between Dataset Characteristics and Structural Bugs

We manually inspected the dataset released by [58] and filtered out the posts related to non-crashing bugs. We found 105 posts in relation to our targeted bugs in DL programs. The DL programs derived from these posts include models with different architectures, such as FCNN and CNN, and different network depths. The distribution of DL models with varying depths obtained from 105 posts is shown in Fig. 2. These posts are manually reviewed

Table 1. Mapping between Different Types of Bugs, Dataset Characteristics, and Verification Rules.

Bug Categories	**Type of Bug**	**Dataset Characteristics used to fix the bug in SO posts**	**Rules**
Activation Function	Wrong type of activation/ Missing/redundant softmax or relu activation	Number of classes Type of problem (regression/classification)	Choice of Non-Linearity (CNL): Checks for missing/redundant/wrong activation function
Layer Properties	Wrong filter for conv layer	Type of images (RGB/Grayscale)	Inaccurate Number of Filters (INF): Checks inappropriate number of filters for each conv layer
	Suboptimal number of neurons	Type of images (RGB/Grayscale)/ Type of problem	Incorrect Number of Neurons (INN): Detects incorrect number of units dense layers
	Wrong amount or type of pooling	Any type of data/problem	Insufficient Downsampling (IDS): Checks for inappropriate amount of pooling after conv layer
Missing Layer	Missing Dropout Layer	Any type of data/problem	Missing or Redundant Dropout (MRD): checks if dropout is applied after dense and conv layer
	Missing Normalization Layer	Any type of data/problem	Missing Normalization Layer (MNL): checks for missing normalization layer after dense and conv layers
Model Properties	Suboptimal Network Architecture	Type of images (RGB/Grayscale)	Inappropriate Number of Convolution Layers (ICL): checks for suboptimal conv layers Improper Number of Fully Connected Layers (IFL): checks for unnecessary dense layers
Preprocessing of Data	Missing Preprocessing	Any type of data/problem	Input Data not Normalized (IDN): checks if the data is normalized or not
Loss Function	Wrong selection of loss function	Number of classes Type of problem (regression/classification)	Labels, output layer activation, and Loss Mismatch (LLM): detects mismatch between output layer activation and loss function
Hyperparameters	Suboptimal Learning Rate	Any type of data/problem	Learning Rate Out-of-Bound (LOB): checks learning rate is in proper range
	Suboptimal Batch Size	Size of training set	Inadequate Batch Size (IBS): checks for inadequate batch size

by authors to understand the debugging process followed by developers to identify the underlying cause of the bug, its symptoms, and the methods used to fix structural bugs. We observe that most of the bugs related to the structure of the model, *e.g.,* wrong activation, and suboptimal neurons which do not cause the program to crash but lead to training issues, can be found at the beginning of the training process using the characteristics of the dataset, *e.g.,* the number of classes and/or type of problem, *i.e.,* regression or classification. Therefore, we mapped each structural bug with the dataset characteristics utilized to fix it. Table 1 shows the mapping between each type of bug and dataset characteristics used to fix the corresponding bug. Bugs and fixes are obtained using the DL models with different architectures, *i.e.,* FCNN and CNN designed for different tasks, such as image classification, text classification, multi-label classification, and regression, and with varying depths (Fig. 2), which highlights the generalizability of using them for bug localization. Below, we discuss the manual labeling process in detail.

Manual Labeling. In the 105 posts obtained after inspecting the dataset of bugs released by Humbatova *et al.* [58], two authors independently reviewed these posts and classified the bugs into various categories following the taxonomy of [58]. Both authors identified seven bug categories: activation function, layer properties, missing layer, model properties, data preprocessing, loss functions, and hyperparameters. After discussion, we found that both authors reached 100% agreement on categorizing these posts into respective categories. The next step is to understand the debugging process followed by the developers to identify the underlying cause of the bug, its symptoms, and the methods used to fix these bugs. During the distribution of the posts into different bug categories, two authors observed certain frequent terms such as type of data, type of task, number of classes, dimension of images, and size of the training dataset in these posts. These terms are used as initial labels to map the bugs and their fixes used in the posts. We followed the procedure described by Biswas *et al.* [44], and two authors (raters) independently labeled these posts. After labeling all posts, we calculated the agreement using Cohen's Kappa coefficient and conducted a discussion session between the raters and moderators (co-authors). We adopted Biswas *et al.*'s [44] interpretation of Kappa ($\kappa = ([0, 1])$, the higher the better). After the first round, we found an almost perfect agreement ($\kappa = 0.85$). There were only a few disagreements about the labeling, which were resolved after a discussion session involving the raters and the moderators. After discussion, all the authors reached a perfect agreement ($\kappa = 1$). Finally, all the authors collectively examined each post for a final pass. The labels after the first round and final labels are provided in our repository [72].

3.3 Verification Rules Creation Methodology

To reaffirm the bug's underlying source and its effect on performance, we reviewed the literature [41, 43, 56, 65–67, 69, 70, 77, 85]. These research papers provide several guidelines and design principles for designing DL programs. We used these guidelines and design principles to define the verification rules. Therefore, for each bug in Table 1, we define a verification rule (discussed in detail in Section 4.1). We also defined the thresholds for various rules using the fix suggestions obtained from 105 posts, illustrated in Section 3.2. Since the fix suggestions were effective for the DL models with different architectures, *i.e.,* FCNN and CNN designed for different tasks, such as image classification, text classification, multi-label classification, and regression, and with varying depths, we utilized them to define the thresholds.

We used the buggy DL programs provided in the defect4ML benchmark [74] to verify these rules. This benchmark has 100 faulty DL programs obtained from *Stack Overflow* and *GitHub* belonging to different bug categories proposed in [58]. We randomly picked one buggy DL program for each bug type supported by Theia and verified the rules. As Theia supports 12 types of bugs (shown in Table 1), we specifically selected 12 programs for this task. This process helped us verify the correctness of defined rules and thresholds.

4 APPROACH

In this section, we first describe the analysis techniques used to detect the structural bugs. Then, we discuss the verification rules and explain how these rules are used in our approach, Theia, to automatically detect and localize structural bugs.

4.1 Detecting Structural Bugs

Context-sensitive analysis is a common interprocedural analysis technique used to develop more efficient programs [83]. Various context-sensitive analysis approaches have been proposed in the past, *e.g.,* call-strings approach [73], functional approach [73], call graphs [54]. Unlike traditional programs where one function can be called multiple times and one function can call another function, DL programs follow different structures where the model is built sequentially by calling different layer APIs, *e.g.,* `Conv2D`, `Activation`, `MaxPooling2D`, provided by DL libraries one after the other. For traditional programs, the call-strings approach is used to keep track of how many times each function is called, how many times it is returned, and what other functions are called by it. In our approach, we used call strings to keep track of API calls and applied verification rules to detect missing, redundant, or wrong API usage. Parameter-sensitive approach [84] is used for traditional programs to analyze each function call independently to determine how the function's parameters affect its functioning. We utilized this approach to examine each API call and its parameters to identify incorrect parameters. Fig. 3 shows an overview of Theia. We built upon the meta-model proposed by Nikanjam *et al.* [76] for DL programs,

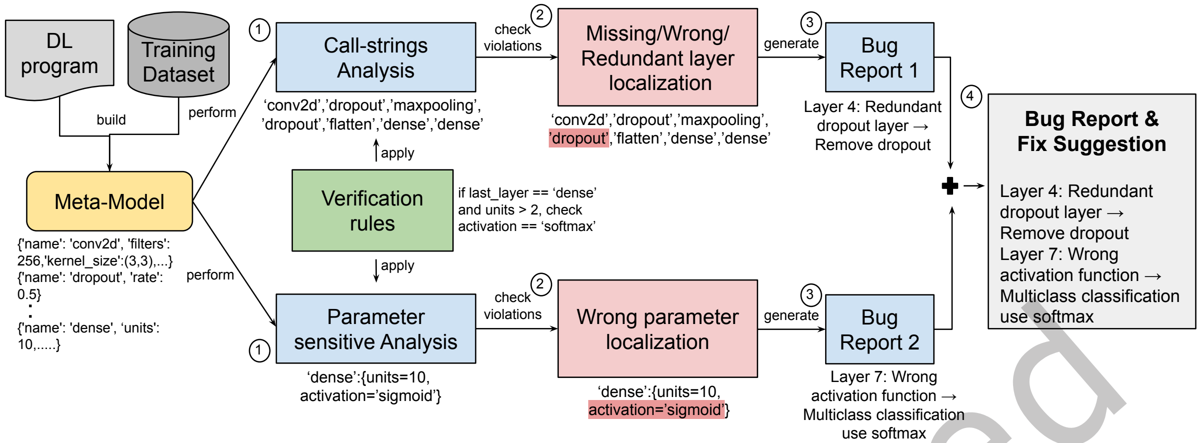


Fig. 3. Overview of Theia.

which captures the various components such as the architecture of the model, learner, and details related to shuffling and batching by parsing the model's source code. However, their meta-model does not capture the characteristics of the data. In our approach, for an executable DL program, a meta-model is built as shown in Fig. 4 which captures the characteristics of the dataset, *e.g.,* type of input, number of classes, and properties of the DL model, *e.g.,* filters in convolution layers, dropout rate. We utilized the `get_config()` and `modules()` APIs provided by *Keras* and *PyTorch* DL libraries, respectively, for parsing the configuration of the model. On the meta-model, call-string analysis and parameter-sensitive analysis are performed and violations are checked using the verification rules discussed below. Violations are used to detect bugs and to keep track of layer numbers which are utilized in bug report for localization. Each analysis generates a bug report that is combined to generate a final bug report with fix suggestions. If Theia detects a bug, the training aborts with a report containing the bug's location and recommended fixes to alert the developer; otherwise, training continues. Below, we discuss the verification rule used by Theia for detecting and localizing each bug shown in Table 1.

4.1.1 Choice of Non-Linearity (CNL). **Rationale:** Convolution and Dense are linear operations; therefore, incorporating non-linear activation functions is crucial in the DL models to satisfy the Universal Approximation Theorem (UAT). Non-linearity is added to the output of convolution and dense layers of DL programs *via* non-linear activation functions, which ensures these models learn from complex data patterns. Activation functions can be saturating (*e.g.,* sigmoid and tanh) or non-saturating (*e.g.,* ReLU and its variants) [75]. These activations transform the value of convolution and dense operation into a restricted range [47]; therefore, applying multiple or redundant activations will result in the wrong output in the last layer. Hence, for each convolution and dense layer activation function is used once. Also, the choice of activation depends on the type of the task, *i.e.,* regression or classification. For example, for image data, Krizhevsky *et al.* [67] have shown the benefit of using non-saturating non-linear activation functions for hidden layers over saturating counterparts. First, non-saturating functions like ReLU help the network to learn faster, thus accelerating the training process. Second, these activation functions help mitigate common training problems, such as exploding and vanishing gradients [87, 89]. Therefore, choosing the right activation function for hidden layers is crucial for enhancing the model's performance.

Detection: If the activation function for hidden layers, *i.e.,* convolution or dense layer is missing or multiple activation functions are used for the same layer, Theia identifies it as a bug. Theia also considers the type of task for which the model is designed, checks incorrect usage of the activation function, and reports it as a bug.

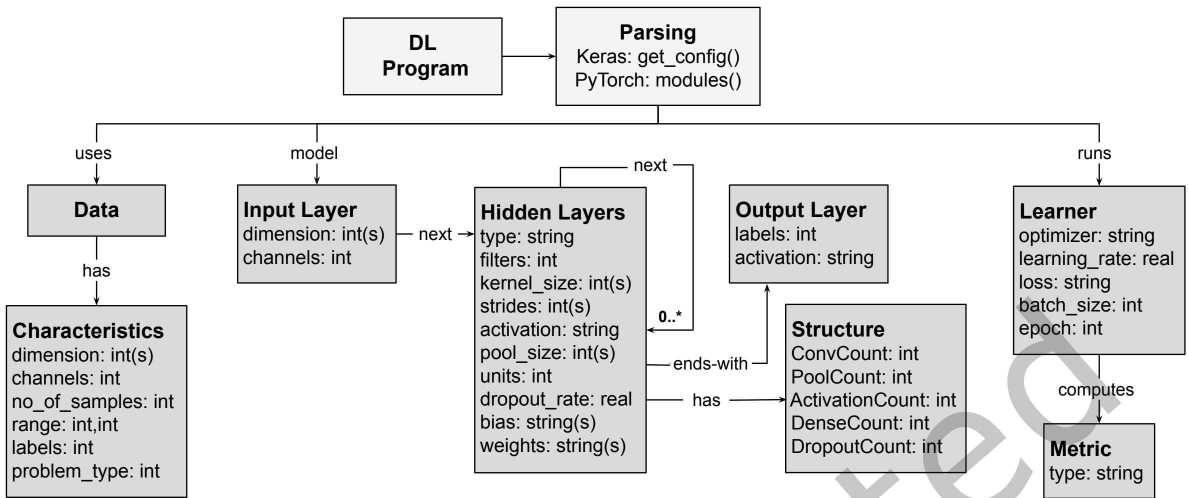


Fig. 4. Meta-Model used in Theia.

4.1.2 Inaccurate Number of Filters (INF). **Rationale:** In CNN programs, features are learned by the convolution layers *via* various filters in each layer. Since the input has multiple features, distinct types of features are learned by these layers, starting from basic features to higher-order features as we go deeper into the network [70]. For example, for image data, the first few layers learn the basic features, *i.e.,* lines, edges, and corners and the deeper layers learn the higher-order features like objects. The filters in each convolution layer depend on the type of input the CNNs are designed for. For instance, if the CNN model is designed for image classification, then the input images can be gray-scale or color. Krizhevsky *et al.* [67] and Simonyan *et al.* [85] have shown the benefit of using more filters for color images as compared to the model designed for gray-scale images [70] as color images have more complex features as compared to the gray-scale images. Convolution layers with too many filters often lead to overfitting of the model on training data, which restricts the model's ability to generalize adequately on test data. While few filters impede the model's capacity to learn, which leads to poor performance during training and testing. While designing CNN programs, the filters in the convolution layer must be configured by the developer by taking into account the training dataset characteristics, *i.e.,* type of data.

Detection: For detecting this bug, Theia checks parameter *filters* in `Conv1D` and `Conv2D` API and considers the dataset characteristic - *channels* captured in meta-model. It checks if the *filters* are less than 16 or more than 512 in each convolution layer for *channels* = 3 (represents color images) or *filters* are less than 6 or more than 256 in each convolution layer for *channels* = 1 (represents gray-scale images or tabular data), the bug is reported with the fix location.

4.1.3 Incorrect Number of Neurons (INN). **Rationale:** The width of the network, *i.e.,* number of neurons in each dense layer of the DL model is defined by considering the task for which the model is built. For instance, for classification tasks, the number of neurons depends on the number of classes for classification [57]. As these neurons learn features during training, a large number of neurons results in more trainable parameters. Therefore, choosing the correct configuration helps in improving performance and results in faster training. Also, for CNNs, the number of neurons in each layer should either remain the same or decrease while moving deeper toward the output layer [67, 70].

Detection: This bug is detected by Theia by checking the *units* in the dense layer excluding the output layer. The *units* must be less than or equal to the size of the input each dense layer receives; otherwise Theia reports it as a bug. For CNN programs, Theia checks if the units in dense layers decreases progressively towards the output layer, reporting a bug if this condition is not met.

4.1.4 Insufficient Downsampling (IDS). **Rationale:** In CNN programs, different filters are used by the convolution layer to generate feature maps [70]. Feature maps extract the position of the features in the input and summarize the presence of features. To make feature maps more robust and make them invariant to the local translation, downsampling is used [70]. Downsampling helps in reducing the size of the feature maps while preserving large or important structural elements. Pooling is the commonly used method for downsampling [67] and it is used after convolution layer(s) to make the model more robust against shifts and distortion [70]. Stacking several convolution layers without using pooling in between makes the model less robust to the local translation and affects the performance of the model. Therefore, to make the model robust, the pooling is recommended to be applied after a stack of few convolution layers [67, 85].

Detection: Theia detects this bug by checking if the pooling layer is missing after 4 consecutive convolution layers.

4.1.5 Missing or Redundant Dropout (MRD). **Rationale:** Dropout is a regularization technique used in DL programs to prevent the model from overfitting and thus helps in better generalization. Srivastava *et al.* [86] proposed this approach and has shown the effectiveness of using dropout on the performance of the DL model. [86] suggests applying dropout after dense and convolution layers once as these layers have learnable parameters.

Detection: To detect this bug, Theia checks if dropout is applied after dense and convolution layers. Theia also counts the number of times dropout (DropoutCount) is applied to each dense and convolution layer. If DropoutCount for each layer is greater than 1, Theia reports it as a bug.

4.1.6 Missing Normalization Layer (MNL). **Rationale:** Batch Normalization is a technique used to train DL model faster. The goal of Batch Normalization is to generate a consistent distribution of activation values throughout the training which helps in faster convergence. Therefore, to train DL models faster, Batch Normalization is recommended after convolution and dense layers before applying non-linearity [82].

Detection: Theia detects this bug by keeping track of layers after dense and convolution layers. If the Batch Normalization layer is missing after these layers and before the activation layer, Theia reports it as a bug.

4.1.7 Inappropriate Number of Convolution Layers (ICL). **Rationale:** In CNN program, the convolution layers are used to extract the local features from the input. The elementary visual features such as edges, lines, *etc.,* are learned by the first few convolution layers [70]. And, subsequent convolution layers are used to learn the higher-order features by combining the features from the previous convolution layers [70]. State-of-the-art CNN architectures [55, 67, 85] showed the advantage of having more convolution layers in the CNN model designed for training datasets with color images. For instance, for color images in ImageNet [48] dataset, popular CNN architectures, *e.g.,* AlexNet [67], VGG [85] used more convolution layers to learn the features in contrast to fewer convolution layers used by LeNet-5 [70] for grayscale images in MNIST [70] dataset. Therefore, the number of convolution layers must be selected by considering the type of images in the training dataset, *i.e.,* grayscale or color.

Detection: This bug is detected by Theia by counting the number of convolution layers. For a dataset with grayscale images, there must be at least 2 or more convolution layers, and for color images, there must be at least 3 or more convolution layers.

4.1.8 Improper Number of Fully Connected Layers (IFL). **Rationale:** In CNN programs, the fully connected layers are used for classification. These layers have a large number of trainable parameters, so more time and memory are required to train them. It is advised [67, 70, 85] to use one or two fully connected layers since they save training time, prevent the model from over-fitting, and improve generalization.

Detection: Theia checks the number of dense layers used in the structure of the CNN model. If the number of dense layers is more than 3, Theia warns the developer to reduce the number of fully connected layers.

4.1.9 Input Data not Normalized (IDN). **Rationale:** Backpropagation is a popular algorithm used to train neural networks [56]. The efficiency of the algorithm depends on the input data [69]. LeCun *et al.* [68] provides several guidelines for more efficient back-propagation. Normalization of the input is one of them. If the input data to the model are close to zero, it results in faster convergence and thus, makes the training faster. Therefore, in the data preprocessing stage, the training data must be normalized in order to achieve better performance.

Detection: Theia detects this bug by checking the range of input values. If the *range* is not between [0,1] or [-1,1], Theia and alerts the developer about it by providing a message in the bug report.

4.1.10 Labels, output layer activation, and Loss Mismatch (LLM). **Rationale:** For image classification, the activation function for the output layer is chosen based on the type of classification, *i.e.,* binary or multi-class classification. And, for regression tasks, for the output layer, linear activation is preferred over non-linear activation [57]. *Loss* is used to evaluate the performance of the model and to compute the error at the time of training. Cross-entropy is the commonly used loss function for classification problems. As suggested by [52, 78], for binary classification, it is preferable to use the sigmoid activation function in the output layer and binary cross-entropy as a loss function to compute the error. For multi-class classification, it is suggested to use the softmax activation function in the output layer. If the loss function is not selected according to the last layer activation function, then due to improper gradient, the model will learn inefficiently.

Detection: Theia uses the problem type passed as input to the callbacks and checks the activation function and loss. If there is any mismatch as explained above, the bug is reported.

4.1.11 Learning Rate Out-of-Bound (LOB). **Rationale:** Learning rate is an important parameter that controls how much the model weights are adjusted with respect to the loss during backpropagation [68]. Too low learning rate increases the training time as the update towards the minima is very small. Sometimes, it is also possible that due to a small learning rate, the training gets stuck on a sub-optimal solution or never converges [89]. And, if the learning rate is set too high, the weight updates will be large which may result in an oscillating loss at the time of training [89].

Detection: Theia looks for an inappropriate learning rate by using the following threshold. If *learning_rate* is greater than 0.01 or *learning_rate* is less than 0.0001, it will be detected as a bug by Theia.

4.1.12 Inadequate Batch Size (IBS). **Rationale:** Batch size is an important training-related hyperparameter whose value impacts the performance of the model [89]. With a smaller batch size, even without looking at the complete training data, the model starts to learn, leading to oscillating loss and it is uncertain that the model will converge to the global optima [89]. A larger batch size, on the other hand, might make the model get trapped in the local minima, which leads to poor generalization and low accuracy [64, 89]. LeCun *et al.* [68] and Bengio *et al.* [43] suggest using 32 as the initial batch size and doubling it until 256.

Detection: Theia detects this bug by checking the batch_size used for training. If the batch size is less than 32 or greater than 256, Theia warns the developer to use the appropriate batch size.

Algorithm 1 gives more details about the steps depicted in Fig. 3. The algorithm first captures the configuration of each layer of the DL model. Then, it iterates through all the layers of the input model and applies the rules, discussed in Section 4.1, according to the type of the layer considering the type of problem and input. For example, for a conv2d layer, the rule `INF()`, is applied considering the type of input. For the pooling layer,`IDS()` is used. As dense layers are used as hidden layers and output layer in the model, the algorithm invokes the rule `INN()` for hidden dense layers, and for the output layer, `LLM()` is applied according to the problem type. Then, the rules

Table 2. Summary of Rules used in Theia.

Bug Type	Model Type	Analysis Technique	API used for Bug Detection	Bug Detection Rule
CNL	FCNN/ CNN	call-strings	Dense(), Conv1d(), Conv2d()	if layer = 'dense' or 'conv' and activation_count = 0 or >1 before next 'dense' or 'pooling' or 'conv' layer
INF	CNN	parameter-sensitive	Conv1d(), Conv2d()	if input_type = 'color_images', conv_filters <16 and conv_filters >512 or if input_type = 'grayscale_images' or 'tabular', conv_filters <6 and conv_filters >256
INN	FCNN/ CNN	parameter-sensitive	Dense()	if dense_layer_units >size of input of each layer
IDS	CNN	call-strings	Conv1d(), Conv2d()	if consecutive_conv_layer_count >4 and layer_next != 'pooling'
MDR	FCNN/ CNN	call-strings	Dense(), Conv1d(), Conv2d()	if layer_hidden = 'activation' and layer_next != 'dropout' or if layer = 'dense' or 'conv' and dropout_count >1 before the next 'dense' or 'pooling' or 'conv' layer
MNL	FCNN/ CNN	call-strings	Dense(), Conv1d(), Conv2d()	if layer = 'dense' or 'conv' and layer_next != 'batch_normalization'
ICL	CNN	call-strings	Conv2d()	if input_type = 'color_images', conv_layer_count <3 or if input_type = 'grayscale_images', conv_layer_count <2
IFL	CNN	call-strings	Dense()	if dense_layer_count >3
IDN	FCNN/ CNN	parameter-sensitive	fit()	if input_range != [0,1] or [-1,1]
LLM	FCNN/ CNN	parameter-sensitive	Dense(), compile()	if problem_type = 'binary_classification', output_layer_activation != 'sigmoid' and loss != 'binary_crossentropy' or if problem_type = 'multiclass_classification', output_layer_activation != 'softmax' and loss != 'categorical_crossentropy' or if problem_type = 'regression', output_layer_activation != 'linear' and loss != 'mse' or 'mae'
LOB	FCNN/ CNN	parameter-sensitive	compile()	if learning_rate >0.01 and learning_rate <0.0001
IBS	FCNN/ CNN	parameter-sensitive	fit()	if batch_size <32 and batch_size >256

`CNL()`, `MRD()`, `MNL()`, which are common for different layers, *i.e.,* conv1d, conv2d, and dense layers are applied. After looping through all the layers of the model, if the model is designed for an image classification task, then the two rules `ICL()` and `IFL()` are applied. Finally, the rules `IBS()` and `LOB()` common for any architecture (FCNN or CNN) are invoked. Each rule localizes different types of bugs discussed in Section 4.1 and records an error message in a list, *bug_report1* or *bug_report2*. The two bug reports, *bug_report1* and *bug_report2* are finally concatenated. If the *final_report* list is not empty, the training process is aborted and the messages, which contain the bug's location and actionable fix, are provided to the user. Otherwise, the algorithm terminates and training starts normally.

The bug detection rules and the analysis techniques used to identify each bug in Theia are summarized in Table 2.

5 EVALUATION

In this section, we discuss the experimental setting and report an empirical evaluation to demonstrate the effectiveness of Theia.

5.1 Research Questions

In this paper, we answer the following research questions.

Algorithm 1: Theia Algorithm

Input : M, problem_type, input_type
Output: Report with bug location and actionable fix

```
bug_report1 ← []
bug_report2 ← []
layer_name ← []
layer_config ← []
for L in M.Layers do
    layer_names.append(L.Name) layer_config.append(L.get_config())
ConvCount = layer_names.count("conv2d")
DenseCount = layer_names.count("dense")
bug_report1.append(IDN(M.input))
for i in range(len(layer_names)) do
    if layer_name[i] == "conv2d" then
        bug_report2.append(INF(input_type, layer_config[i].filters))
    if layer_name[i] == "maxpooling2d" or layer_name[i] "averagepooling2d" then
        bug_report1.append(IDS(ConvCount))
    if layer_name[i] == "dense" then
        if layer_name[i] is the last layer of M then
            bug_report2.append(LLM( problem_type, layer_config[i].activation, M.loss, M.labels))
        else
            bug_report2.append(INN(layer_config[i].units))
    if layer_name[i] == "conv1d" or layer_name[i] == "conv2d" or layer_name[i] == "dense" then
        bug_report2.append(CNL(layer_config[i].activation))
        bug_report1.append(MRD())
        bug_report1.append(MNL())
bug_report1.append(ICL(input_type, ConvCount))
bug_report1.append(IFL(input_type, DenseCount))
bug_report2.append(IBS(M.batch_size))
bug_report2.append(LOB(M.learning_rate))
final_report = bug_report1 + bug_report2
if final_report != null then
    Abort training
    return final_report
```


- **RQ1 (Evaluation):** How effective is Theia in localizing and providing the actionable fixes for bugs in DL programs compared to state-of-the-art?
- **RQ2 (Ablation)**: To what extent does Theia detect each category of bugs correctly?
- **RQ3 (Limitation):** In which cases does our technique fail to detect and localize the bugs?

5.2 Experimental setup

5.2.1 Implementation. We implemented Theia on top of *Keras* 2.3.0 [51], *TensorFlow* 2.1.0 [50], and *PyTorch* 1.13.1 [78]. The meta-model is built by parsing the DL program. The data characteristics are obtained using training data provided as input to the DL program. The configuration of the layers and learner are obtained by using *get_config()* and *modules()* APIs provided by *Keras* and *PyTorch*, respectively. Algorithm 1 is implemented as a Python class which can be imported with Keras/PyTorch program. We conducted all the experiments on a computer with a 4.2 GHz Quad-Core Intel Core i7 processor and 32 GB 2400 MHz DDR4 GB of RAM running the 64-bit MacOS X version 10.15.7.

5.2.2 Benchmark. We collected buggy DL programs developed using *Keras* and *PyTorch* from *Stack Overflow* posts to construct our benchmark. We followed [61] and used the keywords "bug," "`poor performance`," "CNN," "`low accuracy`" to search for posts. We obtained 172 posts. In some posts, we found that pre-trained models provided by deep learning libraries were used. We removed such posts and obtained 63 posts. In most of the posts, the complete code is not provided by the developer. Since Theia needs a complete DL program for evaluation, we considered the posts with full code script. Therefore, we ended up with 40 posts [1–40]. Additionally, we examined the benchmark of NeuraLint [76]. As NeuraLint supports both crash and performance bugs, we filtered out the programs with performance bugs. We obtained 9 programs with bad performance. As Theia needs a DL program with the dataset and DL model for evaluation, we obtained 4 programs (SO# 50079585, 34311586, 51749207, 58844149) with datasets from NeuraLint's benchmark. These programs were already included in our benchmark. We also examined the artifacts provided by the previous works [53, 88]. Due to the overlapping of programs in these benchmarks and 40 programs in our benchmark, we found that these programs were already included in our benchmark (40 posts) during our filtration process. In total, we have 40 buggy DL programs from *Stack Overflow* in our benchmark. The bug dataset contains 22 multi-class classifier models (16 CNNs and 3 FCNNs designed for image data, and 3 FCNNs designed for structural data), 13 binary classifier models (8 CNNs designed for image data, 2 CNNs designed for structural data, and 3 FCNNs designed for structural data), 3 regression models (3 FCNNs designed for structural data) and 1 multi-label classifier (1 FCNNs designed for structural data). The 40 programs in our benchmark are "unseen", *i.e.,* these programs are not considered while determining the thresholds (using 105 programs) in verification rules.

5.2.3 Results Representation. As discussed in Section 1, among the existing approaches for localizing bugs in the DL programs, only NeuraLint detects bugs before training and also supports CNN architecture-related bugs. Therefore, we compared Theia with NeuraLint on 40 buggy DL programs in our benchmark. Table 5 summarizes the results of evaluating Theia and NeuraLint for a multiclass classification task. And, Table 6 summarizes the results of evaluating Theia and NeuraLint for a binary classification and regression task. In both the tables, Table 5 and Table 6, for each DL program, we categorized the buggy programs into different categories indicated by "Bug type" which is obtained by mapping to the bugs specified in our verification rules, "Problem Type" provides the details about the type of task, *i.e.,* regression or classification, "SO #" represents the *Stack Overflow* post #, "Recommended Fix from SO" describes the recommended fix provided by the other users on *Stack Overflow*. The next two columns represent the results of Theia and NeuraLint on our benchmark. The well-known practice is to perform an evaluation on *Stack Overflow* posts with accepted answers [76]. It guarantees the suggested fix is a real fix for the problem and can be used as ground truth for evaluation. In our benchmark, we found 6 posts from *Stack Overflow* without accepted answers. We observed that some of the answers in these posts are marked as useful by users. We considered them as a fix for the problem mentioned in the post. To verify that the recommended fix effectively addressed the issue outlined in the post, we evaluated the performance of the buggy model before and after applying the suggested patch/fix. Loss and accuracy are the common metrics used to evaluate the performance of the DL models. We manually fixed the model following the suggestions from the

Table 3. Performance Comparison of Buggy and Repaired Models Designed for Multiclass Classification Task.

SNo.	SO #	Problem Type	Performance of Buggy Model		Performance After Applying Patch from SO		
			Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)
1	64522751	Multiclass	2.761	82.000	0.298	89.250	7.250 ↑
2	50079585	Multiclass	0.482	76.220	0.511	79.100	2.880 ↑
3	47272383	Multiclass	0.990	64.610	0.480	80.500	15.890 ↑
4	51118032	Multiclass	2.309	9.900	0.599	79.580	69.680 ↑
5	37229086	Multiclass	0.384	86.250	0.314	88.530	2.280 ↑
6	48594888	Multiclass	0.765	73.480	0.424	85.810	12.330 ↑
7	59325381	Multiclass	0.010	9.870	0.050	98.830	88.960 ↑
8	64188884	Multiclass	1.236	48.740	0.899	65.760	17.020 ↑
9	70554413	Multiclass	2.770	6.670	1.072	76.500	69.830 ↑
10	65275387	Multiclass	1.946	14.500	0.413	84.500	70.000 ↑
11	54923573	Multiclass	0.151	93.840	0.160	94.000	0.160 ↑
12	63027146	Multiclass	2.303	9.890	0.253	91.000	81.110 ↑
13	65659888	Multiclass	0.676	49.900	0.076	90.100	40.200 ↑
14	58666904	Multiclass	Crash		0.112	70.100	70.100 ↑
15	55198221	Multiclass	0.998	60.320	0.935	63.840	3.520 ↑
16	55343875	Multiclass	0.064	97.240	0.051	98.290	1.050 ↑
17	38648195	Multiclass	0.201	47.810	0.727	68.190	20.380 ↑
18	48385830	Multiclass	nan	9.870	0.290	91.200	81.330 ↑
19	51930566	Multiclass	0.808	63.330	0.454	91.670	28.340 ↑
20	55328966	Multiclass	nan	9.860	0.008	99.770	89.910 ↑
21	58609115	Multiclass	0.016	99.780	0.070	99.800	0.020 ↑
22	59278771	Multiclass	0.107	50.100	0.059	96.670	46.570 ↑

improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model.
↑ represents increase percentage, ↓ represents decrease percentage.

accepted/useful answers of posts and computed the loss and accuracy before and after applying the fix shown in Table 3 and Table 4. We found that the recommended fix aided in resolving the issue described in the post in 35 out of 40 posts in our benchmark. In 5 out of 40 posts, we observed that the accuracy did not improve much following the fix suggestions from the accepted/useful answers. Therefore, two authors further investigated these posts and found that some fix suggestions are not marked as accepted or useful by developers. However, upon applying these fixes, the two authors found that these patches helped in improving the model's performance. We considered these fix suggestions as correct fixes and included them in the ground truth. These patches led to an average performance improvement of 38% across 40 buggy DL programs. In Table 5 and Table 6, the column labeled "Recommended Fix from SO" serves as the ground truth, which is used to determine the number of true positive and false negative cases. Both the approaches, Theia, and NeuraLint, also detect the bugs that are not recommended by *Stack Overflow* users. For analyzing these results, we adopted the approach used by Nikanjam *et al.* [76]. Two authors independently checked the output and examined the DL program. We found that some structural inefficiencies are not pointed as a fix by any *Stack Overflow* user but are trivial and result in abnormal behavior during training, *e.g.,* multiple activation functions or dropout used for convolution or dense layers, missing pooling layer. For instance, in SO# 47272383, we observed that multiple dropouts are used for the same convolution layer, which is not reported as a fix by the *Stack Overflow* user. The removal of one dropout layer, combined with the suggested fix from the *Stack Overflow* user led to an improvement in the model's performance. We do not consider such fixes as false positives as addressing these structural inefficiencies helps improve the DL program's structure as discussed in Section 4.1 which in turn helps improve the model's performance. Therefore, we have not encountered any false positive cases and do not report them in Table 5 and Table 6. For both the approaches Theia and NeuraLint, the "Yes" indicates whether the bug is identified and localized successfully or not. "–" denotes that the target problem is not yet supported by the approach.

Table 4. Performance Comparison of Buggy and Repaired Models Designed for Binary Classification & Regression Task.

SNo.	SO #	Problem Type	Performance of Buggy Model		Performance After Applying Patch from SO		
			Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)
1	58844149	Binary	7.645	49.860	0.485	76.030	26.170 ↑
2	60261103	Binary	0.894	50.700	0.100	96.400	45.700 ↑
3	56914715	Binary	7.682	49.630	0.490	84.750	35.120 ↑
4	60003876	Binary	Crash		0.938	50.000	50.000 ↑
5	70428592	Binary	Crash		0.000	98.100	98.100 ↑
6	40045159	Binary	0.490	72.880	0.441	80.100	7.220 ↑
7	45378493	Binary	7.620	50.000	0.076	99.000	49.000 ↑
8	51749207	Binary	7.655	49.800	0.011	99.600	49.800 ↑
9	58844149	Binary	7.645	49.860	0.485	76.030	26.170 ↑
10	31880720	Binary	7.660	50.000	0.005	99.900	49.900 ↑
11	39525358	Binary	0.670	61.590	0.575	92.310	30.720 ↑
12	31627380	Binary	9.797	39.040	0.643	68.120	29.080 ↑
13	34673164	Binary	0.128	77.780	0.422	88.890	11.110 ↑
14	34311586	Regression	0.667	33.300	0.684	66.670	33.370 ↑
15	48221692	Regression	2288.030	–	95.283	–	2192.747 ↓
16	48251943	Regression	736.928	–	1.84×10^{-5}	–	736.928 ↓
17	48934338	Regression	1354.247	–	248.703	–	1105.544 ↓
18	44164749	Multi Label classification	nan	29.630	0.449	79.210	49.580 ↑

improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model (for classification).
improvement = loss after fix - loss of buggy model (for regression).
↑ represents increase percentage, ↓ represents decrease percentage.

5.3 RQ1 (Evaluation)

5.3.1 Evaluation on Multiclass Classification Task. We evaluated our approach and compared the state-of-the-art approach, *i.e.,* NeuraLint [76], and reported the results in Table 5. Below, we discuss the different categories of bugs and how NeuraLint performs compared to our approach. For bug type *LLM*, there are 14 programs in Table 5. Theia identified this bug in 14/14 programs (12 Keras and 2 PyTorch programs). To detect this bug, Theia considers the type of problem, *e.g.,* binary classification, and multiclass classification, the last layer activation, and the loss function for which the DL model is built. Whereas, NeuraLint supports this bug type and checks whether the loss function is correctly defined considering the last layer activation function. As discussed in Section 4.1, the last layer activation function and loss functions are defined according to the type of problem, *e.g.,* binary classification, and multiclass classification. As NeuraLint does not consider the type of problem while detecting this bug, it failed to detect bugs in SO # `65275387, 54923573, 55198221, 48385830, 51930566, 55328966, 59278771`. The bugs in SO # `65659888, 58666904` belong to *LLM*, as these are PyTorch programs, NeuraLint does not support PyTorch programs. For programs SO # `37229086, 63027146, 48385830`, our results (in Table 5) show that by considering the characteristics of the dataset, Theia is able to detect all the bugs belonging to different categories *ICL, IDN, INF, LLM, CNL,* whereas, NeuraLint failed to detect all the bugs and detected 2/7 bugs in 3 programs. In total, Theia detected 34/45 bugs found in 22 buggy real-world programs, whereas, NeuraLint detected 13/45 bugs.

5.3.2 Evaluation on Binary Classification & Regression Task. To evaluate the effectiveness of Theia on binary classification and regression tasks, we performed the evaluation on 18 buggy programs obtained from *Stack Overflow* in our benchmark. Table 6 reports the evaluation results of using Theia and NeuraLint on these programs. There are 13 programs for binary classification tasks in Table 6. Most of the programs have *LLM* bugs, Theia successfully detected 12/12 bugs of this category by taking into account the dataset characteristics - number of classes. Whereas, NeuraLint detected 2/12 bugs (SO# 31627380, 34673164) in this category. Both Theia and NeuraLint support CNN program-specific bugs, therefore for SO# 64188884, both the approaches detected bug in *IDS* category. For the regression task, there are 4 programs in Table 6, Theia successfully detected 4/4 bugs

Table 5. Comparison of Bugs Localized by Theia and NeuraLint in Buggy DL Programs Designed for Multiclass Classification Task.

SNo.	Bug Type	Problem Type	SO #	Recommended Fix from SO	Theia			NeuraLint		
					Identify Bug	TP	FN	Identify Bug	TP	FN
1	LLM	Multiclass	64522751	Change loss function	Yes	2	0	Yes	1	1
	LOB			Reduce the learning rate	Yes			–		
2	LLM	Multiclass	50079585	Change last layer activation function	Yes	2	0	Yes	2	0
	LLM			Change loss function	Yes			Yes		
3	ICL	Multiclass	47272383	Increase network depth	Yes	1	1	–	0	2
	–			Increase dataset size	–			–		
4	LOB	Multiclass	51118032	Reduce the learning rate	Yes	1	0	–	0	1
5	ICL	Multiclass	37229086	Increase network depth	Yes	1	0	–	0	1
6	–	Multiclass	48594888	Change weight initializer	–	1	3	–	2	3
	INF			Increase convolution layer filters while going deeper	Yes			Yes		
	–			Change kernel size	–			Yes		
	–			Decrease dropout rate	–			–		
7	IDN	Multiclass	59325381	Normalize the test data	Yes	1	0	–	0	1
8	–	Multiclass	64188884	Reduce Dropout layers	No	1	1	–	1	1
	IDS			Add pooling layer	Yes			Yes		
9	–	Multiclass	70554413	Increase dataset size and randomize the data	–	1	1	–	0	2
	ICL			Improve network design	Yes			–		
10	CNL	Multiclass	65275387	Change Dense layer activation	Yes	2	2	–	0	4
	LLM			Change last layer activation function	Yes			No		
	–			Use data augmentation	–			–		
	–			Remove input_shape from all layers except input layer	–			–		
11	LLM	Multiclass	54923573	Change last layer activation function	Yes	1	0	No	0	1
12	IDN	Multiclass	63027146	Normalize train and test data	Yes	3	0	–	1	2
	ICL			Increase network depth	Yes			–		
	INF			Increase convolution layer filters while going deeper	Yes			Yes		
13	LLM	Multiclass	65659888	Remove last softmax activation layer	Yes	1	0	–	–	–
14	LLM	Multiclass	58666904	Remove last layer activation	Yes	2	0	–	–	–
	LLM			Change loss function	Yes			–		
15	LLM	Multiclass	55198221	Change last layer activation function	Yes	1	0	No	0	1
16	LLM	Multiclass	55343875	Change last layer activation function	Yes	2	0	Yes	2	0
	LLM			Change loss function	Yes			Yes		
17	LLM	Multiclass	38648195	Change loss function	Yes	1	1	Yes	1	0
	–			Change optimizer	–			–		
18	CNL	Multiclass	48385830	Add activation function in hidden layers	Yes	3	0	Yes	1	2
	CNL			Add last layer activation function	Yes			No		
	LLM			Change loss function	Yes			No		
19	LLM	Multiclass	51930566	Change last layer activation function	Yes	1	0	No	0	1
20	LLM	Multiclass	55328966	Change last layer activation function	Yes	3	0	No	0	3
	IDN			Normalize the data	Yes			–		
	CNL			Add activation for first dense layer	Yes			No		
21	LLM	Multiclass	58609115	Change last layer activation function	Yes	2	0	Yes	2	0
	LLM			Change loss function	Yes			Yes		
22	IFL	Multiclass	59278771	Add more dense layer	No	1	2	–	0	3
	INN			Increase units in dense layers	No			–		
	LLM			Change last layer activation function	Yes			No		
				Total		**34**	**11**		**13**	**29**

belonging to *LLM, LOB* bug categories. On the other hand, NeuraLint was not able to find any of these bugs. There is 1 program for the multi-label classification task, SO# 44164749, both Theia and NeuraLint failed to detect the bug in this program. We investigated the reason for it and found that for multi label classification, the mapping between the last layer activation function and loss function is different than the multiclass classification as discussed in 4.1. Therefore, in the verification rule, *LLM*, there is a need to add a proper mapping.

Table 6. Comparison of Bugs Localized by Theia and NeuraLint in Buggy DL Programs Designed for Binary Classification & Regression Task.

SNo.	Bug Type	Problem Type	SO #	Recommended Fix from SO	Theia			NeuraLint		
					Identify Bug	TP	FN	Identify Bug	TP	FN
1	LLM	Binary	58844149	Change last layer activation function	Yes	1	0	No	0	1
2	LOB	Binary	60261103	Reduce the learning rate	Yes	1	0	–	0	1
3	LLM	Binary	56914715	Change last layer activation function	Yes	1	0	No	0	1
4	LLM	Binary	60003876	Change loss function	Yes	1	1	–	–	–
	–			Initialize optimizer to zero gradient	–			–		
5	LLM	Binary	70428592	Remove last softmax activation layer	Yes	2	0	–	–	–
	LLM			Use proper loss fucntion	Yes			–		
6	ICL	Binary	40045159	Increase network depth	Yes	2	0	–	1	1
	INF			Increase number of conv filters	Yes			Yes		
7	LLM	Binary	45378493	Change last layer activation function	Yes	1	0	No	0	1
8	LLM	Binary	51749207	Change last layer activation function	Yes	1	0	No	0	1
9	LLM	Binary	58844149	Change last layer activation function	Yes	1	0	No	0	1
10	LLM	Binary	31880720	Change last layer activation function	Yes	1	0	No	0	1
11	IDN	Binary	39525358	Normalize the data	Yes	2	1	–	0	3
	–			Increase number of epochs	–			–		
	MRD			Add dropout layers after hidden layers	Yes			–		
12	MRD	Binary	31627380	Add dropout layers	No	2	1	–	2	1
	LLM			Change last layer activation function	Yes			Yes		
	LLM			Change loss function	Yes			Yes		
13	LLM	Binary	34673164	Change loss function	Yes	3	2	Yes	1	4
	–			Change optimizer	–			–		
	MNL			Add Batch Normalization	Yes			–		
	CNL			Change hidden layer activation function	No			–		
	IDN			Normalize the data	Yes			–		
14	LLM	Regression	34311586	Remove last softmax activation layer	Yes	1	0	No	0	1
15	LLM	Regression	48221692	Remove last layer activation	Yes	1	0	No	0	1
16	LLM	Regression	48251943	Remove last layer activation function	Yes	1	0	No	0	1
17	LOB	Regression	48934338	Reduce learning rate	Yes	1	0	–	0	1
18	LLM	Multi Label classification	44164749	Change last layer activation function	No	0	2	No	0	2
	LLM			Change loss function	No			No		
Total						**23**	**7**		**4**	**22**

5.3.3 Evaluation of Actionable Fixes on Buggy DL Programs. We investigated the impact of the fix suggestions provided by Theia on improving the performance of the buggy DL program after repair. To investigate this, the two authors manually addressed the bugs for each of the 40 programs in our benchmark, following the line numbers and fix recommendations from Theia and NeuraLint, and compared the results with the performance of the original buggy model. If the fix does not improve the performance of the buggy model, we consider the fix suggestions as false alarms (FP). Theia localizes the bug and provides developer hints at the potential solutions, whereas NeuraLint identifies the bug but does not provide guidance on potential solutions. For instance, for inappropriate loss function, Theia provides the message: "Change loss function –> Use categorical_crossentropy", whereas, NeuraLint provides the message: "Learner ==> The loss should be correctly defined and connected to the layer in accordance with its input conditions (*i.e.,* shape and type)-post_activation". After fixing the bug, we rerun Theia and NeuraLint on the modified DL program and repeat the process until no bugs are reported by both tools. The comparison of loss and accuracy of the original buggy program and the manually repaired program using actionable fixes from Theia and NeuraLint are reported in Table 7 and Table 8. The results show that Theia successfully provided actionable fixes, resulting in an average performance enhancement by 41% in 34 out of 40

Table 7. Comparison of Impact of Actionable Fixes by Theia and NeuraLint on Buggy Models Performance Designed for Multiclass Classification Task.

SNo.	SO #	Performance of Buggy Model		Performance After Fix							
				NeuraLint				Theia			
		Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)	FP	Loss	Accuracy (in %)	Improvement (in %)	FP
1	64522751	2.761	82.000	2.327	9.780	-72.220 ↓	Yes	0.510	82.190	0.190 ↑	No
2	50079585	0.482	76.220	0.841	64.380	-11.840 ↓	Yes	0.508	80.890	4.670 ↑	No
3	47272383	0.990	64.610	1.776	19.810	-44.800 ↓	Yes	0.380	88.420	23.810 ↑	No
4	51118032	2.309	9.900	2.309	9.900	0.000 →	No	0.509	82.020	72.120 ↑	No
5	37229086	0.384	86.250	0.107	96.370	10.120 ↑	No	0.616	78.560	-7.690 ↓	Yes
6	48594888	0.765	73.480	0.772	73.160	-0.320 ↓	Yes	0.571	79.920	6.440↑	No
7	59325381	0.010	9.870	0.040	9.870	0.000 →	No	0.023	99.340	89.470 ↑	No
8	64188884	1.236	48.740	1.111	54.000	5.260 ↑	No	0.946	62.590	13.850 ↑	No
9	70554413	2.770	6.670	2.770	6.670	0.000 →	No	1.073	77.440	70.770 ↑	No
10	65275387	1.946	14.500	0.411	85.710	71.210 ↑	No	0.076	98.400	83.900 ↑	No
11	54923573	0.151	93.840	0.177	92.770	-1.070 ↓	Yes	0.916	92.860	-0.980 ↓	Yes
12	63027146	2.303	9.890	0.746	74.240	64.350 ↑	No	0.721	74.920	65.030 ↑	No
13	65659888	0.676	49.900	–	–	–	–	0.007	93.010	43.110 ↑	No
14	58666904	Crash		–	–	–	–	0.114	72.500	72.500 ↑	No
15	55198221	0.998	60.320	0.960	62.290	1.970 ↑	No	0.710	72.320	12.000 ↑	No
16	55343875	0.064	97.240	0.039	98.400	1.160 ↑	No	0.064	97.800	0.560 ↑	No
17	38648195	0.201	47.810	0.839	59.030	11.220 ↑	No	0.886	58.030	10.220 ↑	No
18	48385830	nan	9.870	0.094	10.110	0.240 ↑	No	1.086	63.530	53.660 ↑	No
19	51930566	0.808	63.330	0.439	76.890	13.560 ↑	No	0.811	73.330	10.000 ↑	No
20	55328966	nan	9.860	0.136	95.510	85.650 ↑	No	0.040	98.710	88.850 ↑	No
21	58609115	0.016	99.780	0.652	0.340	-99.440 ↓	Yes	0.623	0.400	-99.380 ↓	Yes
22	59278771	0.107	97.330	0.233	87.560	-9.770 ↓	Yes	0.477	88.000	-9.330 ↓	Yes

The highlighted rows indicate programs where Theia's fix suggestions did not improve model performance.
improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model.
↑ represents increase percentage, ↓ represents decrease percentage, → represents no change, and – indicates the model not supported yet.

buggy DL programs. In contrast, the fix suggestions from NeuraLint led to an average performance improvement of 30% in 19 out of 40 programs. This highlights the effectiveness of our approach in detecting structural bugs that lead to suboptimal performance during training.

5.3.4 Evaluation on Normal Programs. We conducted a more thorough investigation into the effects of applying Theia on normal programs, aiming to investigate any instances of false alarms in these programs. As shown in Table 3 and Table 4, the patches/fixes suggested by *Stack Overflow* users successfully resolved the bugs present in all 40 programs in our benchmark, resulting in improved performance. Therefore, we utilized these patches to create a benchmark of 40 normal programs. These programs are available in our repository [72]. We evaluated the impact of fix suggestions provided by Theia on these 40 normal programs. We followed the same procedure as described in Section 5.3.3. Two authors manually addressed the bugs for each of the 40 normal programs, following the line numbers and fix recommendations from Theia and NeuraLint and compared it with the performance of the normal model. If the fix does not improve the performance of the model, we consider the fix suggestions as false alarms (FP). The impact on the performance of the normal programs after applying patches is shown in Table 9 and Table 10. On normal programs both Theia (average performance improvement of 6%) and NeuraLint (average performance improvement of 4%) demonstrated a comparable performance on 40 programs, leading to performance improvements in 28 out of the 40 normal programs. Both tools negatively impacted the performance of 12 programs, resulting in 12 false alarms (FP). We investigated the reason for false alarms produced by Theia in 12 out of 40 programs. Theia suggests to add Batch Normalization and Dropout layers after convolution and dense layers (Rules - *MNL* and *MRD*). We observed that for less complex models, the addition of these layers after

Table 8. Comparison of Impact of Actionable Fixes by Theia and NeuraLint on Buggy Models Performance Designed for Binary Classification & Regression Task.

SNo.	SO #	Performance of Buggy Model		Performance After Fix							
				NeuraLint				Theia			
		Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)	FP	Loss	Accuracy (in %)	Improvement (in %)	FP
1	58844149	7.645	49.860	0.359	83.710	33.850 ↑	No	0.398	81.520	31.660 ↑	No
2	60261103	0.894	50.700	0.723	49.600	-1.100 ↓	Yes	0.752	50.900	0.200 ↑	No
3	56914715	7.682	49.630	0.607	74.250	24.620 ↑	No	0.136	95.380	45.750 ↑	No
4	60003876	Crash		–	–	–	–	0.016	91.700	91.700 ↑	No
5	70428592	Crash		–	–	–	–	0.003	98.600	98.600 ↑	No
6	40045159	0.490	72.880	0.485	76.790	3.910 ↑	No	0.485	76.750	3.870 ↑	No
7	45378493	7.620	50.000	0.116	97.000	47.000 ↑	No	0.073	97.000	47.000 ↑	No
8	51749207	7.655	49.800	0.008	99.000	49.200 ↑	No	0.031	99.000	49.200 ↑	No
9	58844149	7.645	49.860	0.359	83.710	33.850 ↑	No	0.398	81.520	31.660 ↑	No
10	31880720	7.660	50.000	0.003	99.000	49.000 ↑	No	0.004	99.810	49.810 ↑	No
11	39525358	0.670	61.590	0.670	61.590	0.000 →	No	0.543	92.310	30.720 ↑	No
12	31627380	9.797	39.040	0.640	68.260	29.220 ↑	No	0.428	81.180	42.140 ↑	No
13	34673164	0.128	77.780	0.776	77.780	0.000 →	No	0.450	77.780	0.000 →	No
14	34311586	0.667	33.300	0.689	66.670	33.370 ↑	No	0.686	66.670	33.370 ↑	No
15	48221692	2288.030	–	-68722.021	–	71010.051 ↑	Yes	916.983	–	1371.047 ↓	No
16	48251943	736.928	–	736.928	–	0.000 →	No	131.660	–	605.268 ↓	No
17	48934338	1354.247	–	1354.247	–	0.000 →	No	42.927	–	1311.320 ↓	No
18	44164749	nan	29.630	nan	29.630	0.000 →	No	nan	29.630	0.000 →	No

The highlighted rows indicate programs where Theia's fix suggestions did not improve model performance.
improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model (for classification).
improvement = loss after fix - loss of buggy model (for regression).
↑ represents increase percentage, ↓ represents decrease percentage, → represents no change, and – indicates the model not supported yet.

each convolution and dense layer leads to excessive regularization, thereby compromising the performance of these models.

5.4 RQ2 (Ablation)

Table 11 shows the performance of Theia on different types of bugs found in DL programs in our benchmark. *LLM* is the most prevalent bug type occurring in real-world buggy programs obtained from *Stack Overflow* in our benchmark. Theia successfully detected 33/35 bugs in this category. Whereas, NeuraLint successfully detected 11/35 bugs of this category. For the second-most prevalent bug type *CNL*, Theia correctly identified 4/5 bugs, and, NeuraLint detected 1/5 bugs. For bugs specific to CNN programs, *INF* and *IDS*, both Theia and NeuraLint were able to detect all the bugs of these categories. There are 12 bugs represented by the "Other" column in Table 11 which are not supported by both Theia and NeuraLint. Theia detected 57/75 bugs from different categories, while NeuraLint detects 17/75 bugs in these real-world buggy programs.

5.5 RQ3 (Limitation)

The scope of Theia is defined as FCNN and CNN programs designed for regression and classification tasks using two deep-learning libraries, *Keras* and *PyTorch*. Other architectures like Recurrent Neural Networks (RNNs) or pretrained DL models are not supported by Theia. Theia can be extended to support other architectures by adding new rules specific to those architectures. Theia is designed to detect 12 structural bugs; therefore, as shown in Table 11 (Bug Categories - Other), it failed to find bugs due to insufficient data, wrong optimizer, incorrect weight initializer, epochs, and dropout rate. As Theia detects bugs at the beginning of the training, some of these bugs, *e.g.,* insufficient data cannot be detected before training. Similarly, different optimizers have different convergence rates which cannot be determined at the early stage of training. Identifying such bugs is a limitation

Table 9. Comparison of Impact of Actionable Fixes by Theia and NeuraLint on Normal Models Performance Designed for Multiclass Classification Task.

SNo.	SO #	Performance of Normal Model		Performance After Fix							
				NeuraLint				Theia			
		Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)	FP	Loss	Accuracy (in %)	Improvement (in %)	FP
1	64522751	0.298	89.250	0.282	89.740	0.490 ↑	No	0.492	82.920	-6.330 ↓	Yes
2	50079585	0.511	79.100	0.816	63.240	-15.860 ↓	Yes	0.526	79.830	0.730 ↑	No
3	47272383	0.480	79.600	1.763	20.450	-59.150 ↓	Yes	0.392	87.340	7.740 ↑	No
4	51118032	0.599	79.580	0.599	79.580	0.000 →	No	0.315	88.690	9.110 ↑	No
5	37229086	0.314	88.530	0.237	91.550	3.020 ↑	No	0.429	84.790	-3.740 ↓	Yes
6	48594888	0.424	85.810	0.424	85.810	0.000 →	No	0.333	88.270	2.460	No
7	59325381	0.050	98.830	0.040	98.950	0.120 ↑	No	0.024	99.270	0.440 ↑	No
8	64188884	0.899	65.760	0.899	65.760	0.000 →	No	0.760	71.440	5.680 ↑	No
9	70554413	1.072	76.500	2.770	7.440	-69.060 ↓	Yes	1.098	78.330	1.830 ↑	No
10	65275387	1.946	84.500	1.854	16.700	-67.800 ↓	Yes	1.854	20.000	-64.500 ↓	Yes
11	54923573	0.160	94.000	0.233	90.150	-3.850 ↓	Yes	0.194	94.100	0.100 ↑	No
12	63027146	0.253	91.000	0.414	85.390	-5.610 ↓	Yes	0.495	82.840	-8.160 ↓	Yes
13	65659888	0.076	90.100	–	–	–	–	0.004	94.410	4.310 ↑	No
14	58666904	0.112	70.100	–	–	–	–	0.114	72.500	2.400 ↑	No
15	55198221	0.935	63.840	0.935	63.840	0.000 →	No	0.807	68.570	4.730 ↑	No
16	55343875	0.051	98.290	0.051	98.290	0.000 →	No	0.022	99.760	1.470 ↑	No
17	38648195	0.727	68.190	0.727	68.190	0.000 →	No	0.728	68.400	0.210 ↑	No
18	48385830	0.290	91.200	0.290	91.200	0.000 →	No	0.723	76.760	-14.440 ↓	Yes
19	51930566	0.454	91.670	0.454	91.670	0.000 →	No	0.698	81.670	-10.000 ↓	Yes
20	55328966	0.008	99.770	0.008	99.770	0.000 →	No	0.030	99.770	0.000 →	No
21	58609115	0.070	99.800	0.776	0.170	-99.630 ↓	Yes	0.633	0.500	-99.300 ↓	Yes
22	59278771	0.059	96.670	0.059	96.670	0.000 →	No	0.106	96.670	0.000	No

The highlighted rows indicate the false positives reported by Theia.
improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model (for classification).
improvement = loss after fix - loss of buggy model (for regression).
↑ represents increase percentage, ↓ represents decrease percentage, → represents no change, and – indicates the model not supported yet.

of our approach. Therefore, Theia failed to detect 12 bugs in our benchmark. We aim to address these bugs by integrating training monitoring into Theia in the future.

5.6 Result and Discussion

Our technique, Theia focuses on identifying structural defects, which are mainly caused by mistakes made by developers during the design of DL programs. These design mistakes may have severe consequences which lead to incorrect output or poor generalization after training the DL model. Detecting these flaws at an early stage of the training process has potential to save computational resources and the developer's time. The results show that Theia outperforms state-of-the-art NeuraLint. Specifically, for real-world programs from *Stack Overflow*, Theia identified and localized 34/45 bugs found in 22 buggy DL programs designed for multiclass classification tasks. However, NeuraLint detected 13/45 bugs in 22 buggy DL programs. For binary classification and regression tasks, Theia detected and localized 23/30 bugs found in 18 buggy DL programs, whereas, NeuraLint identified 4/30 bugs. In total, Theia successfully detected 57/75 bugs in 40 real-world buggy programs obtained from *Stack Overflow*. However, Theia failed to detect 12 bugs from our benchmark as these bugs provide symptoms during training and Theia does not support them. We plan to investigate these bugs in our future work.

Table 10. Comparison of Impact of Actionable Fixes by Theia and NeuraLint on Normal Models Performance Designed for Binary Classification & Regression Task.

SNo.	SO #	Performance of Normal Model		Performance After Fix							
				NeuraLint				Theia			
		Loss	Accuracy (in %)	Loss	Accuracy (in %)	Improvement (in %)	FP	Loss	Accuracy (in %)	Improvement (in %)	FP
1	58844149	0.485	76.030	0.350	84.210	8.180 ↑	No	0.392	81.910	5.880 ↑	No
2	60261103	0.100	96.400	0.732	51.100	-45.300 ↓	Yes	0.755	52.100	-44.300 ↓	Yes
3	56914715	0.490	84.750	0.490	84.750	0.000 →	No	0.154	95.880	11.130 ↑	No
4	60003876	0.938	50.000	–	–	–	–	0.014	92.100	42.100 ↑	No
5	70428592	0.000	98.100	–	–	–	–	0.494	81.200	-16.900 ↓	Yes
6	40045159	0.441	80.100	0.441	80.100	0.000 →	No	0.467	80.400	0.300 ↑	No
7	45378493	0.076	99.000	0.113	98.000	-1.000 ↓	Yes	0.069	99.100	0.100 ↑	No
8	51749207	0.011	99.600	0.011	99.600	0.000 →	No	0.013	99.600	0.000 →	No
9	58844149	0.485	76.030	0.350	84.210	8.180 ↑	No	0.392	81.910	5.880 ↑	No
10	31880720	0.005	99.900	0.003	99.000	-0.900 ↓	Yes	0.027	99.900	0.000 →	No
11	39525358	0.575	92.310	0.575	92.310	0.000 →	No	0.527	99.000	6.690 ↑	No
12	31627380	0.643	68.120	0.643	68.120	0.000 →	No	0.423	82.580	14.460 ↑	No
13	34673164	0.422	88.890	3.931	77.780	-11.110 ↓	Yes	0.479	88.900	0.010 ↑	No
14	34311586	0.684	66.670	0.679	66.670	0.000 →	No	0.680	66.670	0.000 →	No
15	48221692	95.283	–	95.283	–	0.000 →	No	917.720	–	-822.437 ↑	Yes
16	48251943	0.000018446	–	0.000018446	–	0.000 →	No	131.660	–	-131.660 ↑	Yes
17	48934338	248.703	–	248.703	–	0.000 →	No	76.370	–	172.333 ↓	No
18	44164749	0.449	79.210	nan	29.630	-49.580 ↓	Yes	nan	29.630	-49.580 ↓	Yes

The highlighted rows indicate the false positives reported by Theia.
improvement % = accuracy (in %) after fix - accuracy (in %) of buggy model (for classification).
improvement = loss after fix - loss of buggy model (for regression).
↑ represents increase percentage, ↓ represents decrease percentage, → represents no change, and – indicates the model not supported yet.

Table 11. Comparison of Bugs Localized by Theia and NeuraLint Across Different Bug Categories.

Bug Categories	Total Bugs	Theia	NeuraLint
LLM	35	33	11
CNL	5	4	1
ICL	5	5	–
IDN	5	5	–
LOB	4	4	–
INF	3	3	3
MRD	2	1	–
INN	1	0	–
IDS	1	1	1
MNL	1	1	–
IFL	1	0	–
IBS	0	0	–
Other	12	–	–

6 THREATS TO VALIDITY

External Threat: We meticulously selected 105 posts from the dataset provided by [58] to understand the mapping between different types of bugs and dataset characteristics used in these posts to fix the bug. To enhance the generalizability of our research for future work, we propose incorporating additional sources, such as GitHub, to validate the applicability of the proposed approach across a broader range of real-world use cases. Additionally, the design of our verification rules was influenced by insights from the literature [41, 43, 56, 65–67, 69, 70, 77, 85], which could impact the generalizability of the study. To mitigate potential biases, we utilize the defect4ML benchmark, which comprises 100 buggy DL programs collected from *Stack Overflow* and *GitHub*, encompassing various bug categories. This benchmark serves as a reliable means to evaluate our proposed methodology. We acknowledge that the conclusions drawn from this study provide an initial exploration of the bug categories and the challenges DL developers face in addressing these issues. To evaluate our approach, we considered "Recommended Fix from SO" as ground truth. To mitigate the bias due to the selection of the fixes as the ground truth, we applied the patch/fix to the buggy models and evaluated the model's performance before and after the fix. Also, to mitigate the bias due to randomness in DNN models, we ran each program three times and compared the average accuracy of both the buggy and repaired programs. We observed that these patches improved the performance of all the 40 models.

Internal Threat: We were primarily concerned about the implementation of our verification rules. Each rule requires to exact different layers of the model in sequence. To mitigate this threat, after designing and implementing Theia, the authors carefully reviewed the code to reduce the chances of errors. We evaluate our approach, Theia on 40 buggy DL programs. We considered "`Recommended Fix from SO`" as ground truth to evaluate our approach. As Theia detects some bugs that were not specified in the ground truth, we need to verify the bugs reported are not false positive. To mitigate this threat, we verified the correctness of these bugs. Two authors independently examined the output generated by Theia. They fixed the bugs using the actionable fixes reported by Theia and checked the accuracy before and after the fix. If the model's performance is improved, the reported bugs are not considered false positives.

7 RELATED WORK

7.1 Empirical study on Deep Learning Bugs

In recent years, several empirical studies have investigated types of bugs in DL programs [58, 61, 90]. These studies have examined the symptoms and root causes of the deep learning bugs using the *Stack Overflow* posts and GitHub commits. Zhang *et al.* [90] have studied the TensorFlow program bugs and identified 4 symptoms and 7 root causes for these bugs. Meanwhile, Islam *et al.* [61] studied real-world bugs in programs based on five deep-learning libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *PyTorch*, and identified 5 types of bugs and 10 root causes for these bugs. They have also studied the impacts of these bugs on DL programs. Another study was conducted by Islam *et al.* [62] to understand the bug fix patterns in DL programs and the challenges and risks involved in fixing them. The study finds that bug localization and fixing is very difficult in DL programs as fixing one bug may introduce new bugs in the code. Humbatova *et al.* [58] has provided a taxonomy of real faults in Deep Learning Systems. The faults are divided into 5 broad categories. Their study states that the faults in `Model` and `Training` categories mostly lead to performance-related issues. whereas faults in the other three categories "GPU Usage", "API" and "Tensors and Inputs" leads to a crash. Cao *et al.* [45] conducted the first comprehensive study to characterize performance problems in the Deep learning systems designed using TensorFlow and Keras. However, this work focused on the impact of time and resources (*e.g.,* GPU memory and power), whereas our work emphasizes localizing the structural bugs in FCNN and CNN models by analyzing the characteristics of datasets in real-world models written in *PyTorch* and *Keras*.

7.2 Fault localization for Deep Learning Programs

Due to the reliability on a lot of hyperparameters, the bugs in DL programs are different from the traditional software programs. As the traditional fault localization techniques cannot be applied directly to DL programs which drew the researcher's attention to develop new techniques for fault localization in DL programs. Therefore, various approaches have been proposed in the past for automatically detecting, localizing, and repairing DL program bugs. Nikanjam *et al.* [76] proposed NeuraLint, a static analysis approach for automatic fault detection in deep learning programs. NeuraLint identifies the root cause of the bug based on pre-defined verification rules and also provides a message suggesting how to fix the bug. Although NeuraLint can detect bugs in FCNN models and is also capable of detecting bugs specific to CNN architecture, the goals of NeuraLint and Theia are the same. However, Theia considers the characteristics of the training dataset to automatically detect bugs, which allows it to outperform NeuraLint by detecting more structural bugs. Schoop *et al.* [81] proposed UMLAUT, which debugs DL programs using program structure and model behavior. Eniser *et al.* [49] proposed DeepFault, which identifies suspicious neurons for fault localization in DL programs. Wardat *et al.* [88] propose DeepLocalize, a dynamic fault localization technique for DL programs. DeepDiagnosis [87] is another dynamic fault localization technique that detects various symptoms during training and provides actionable fixes. A learning-based fault diagnosis and localization approach DeepFD is proposed by Cao *et al.* [46] which maps fault localization tasks to a learning problem. Braiek *et al.* [42] proposed a property-based debugging approach that detects bugs in three phases, *i.e.,* pre-training, during training, and post-fitting. Although [42, 46, 49, 81, 87, 88] can detect bugs in DL programs, however, these approaches do not support CNN-architecture-specific bugs. Ghanbari *et al.* [53] proposed a mutation-based fault localization approach for DL programs in which the mutants of pre-trained model are created to detect the bugs in DL programs. Despite supporting faults related to CNN architecture, such as strides and filters in the convolution layer, it discovers bugs post-training. In contrast to these dynamic fault localization approaches, our approach, Theia, works at the beginning of the training process and identifies the inappropriate configurations that results in faulty behavior during training. This makes Theia significantly faster than these approaches.

8 CONCLUSIONS AND FUTURE WORK

We propose an approach, named Theia, to automatically detect 12 structural bugs in DL programs designed using two deep learning libraries, *Keras* and *PyTorch*. We considered the characteristics of the training dataset and defined verification rules to localize them. Theia utilizes these rules to detect the bugs, localize them, and alert the developer with an informative message containing actionable fixes in buggy DL programs. The bug's location and descriptive message help the developer easily locate the bug and improve the structure of the DL program. Theia performs bug localization at the beginning of the training process, thereby saving the time and computational resources of the developer. Theia outperforms state-of-the-art NeuraLint by localizing and suggesting the correct fixes for 57/75 buggy programs in our benchmark. In the future, we plan to expand Theia to support other architectures like RNNs.

9 DATA AVAILABILITY

The benchmark consisting of 40 buggy DL programs obtained from *Stack Overflow*, files associated with our manual labeling process, and source code of Theia are available in this repository [72] which allows other researchers to reproduce the results for future research.

10 ACKNOWLEDGMENTS

The authors thank the anonymous ICSE 2024 and TOSEM reviewers for their valuable feedback and Ali Ghanbari for his insightful comments. This work is supported by the US National Science Foundation (NSF) under grants #2512857, #2512858, CCF-15-18897, CNS-15-13263, CNS-21- 20448, CCF-19-34884, CCF-22-23812, and NRT-21-52117, the Fonds de Recherche du Quebec (FRQ), the Canadian Institute for Advanced Research (CIFAR), and the Natural Sciences and Engineering Research Council of Canada (NSERC). All opinions are those of the authors and do not reflect the views of the sponsors.

REFERENCES

[1] 2015. CCannot train a neural network solving XOR mapping. https://stackoverflow.com/questions/34311586/.
[2] 2015. How to prepare a dataset for Keras? https://stackoverflow.com/questions/31880720/.

[3] 2015. Trying Kaggle Titanic with keras .. getting loss and valid_loss -0.0000. https://stackoverflow.com/questions/31627380/.
[4] 2016. Accuracy not high enough for dogs_cats classification dataset using CNN with Keras-Tf python. https://stackoverflow.com/questions/40045159/.
[5] 2016. How to train and tune an artificial multilayer perceptron neural network using Keras? https://stackoverflow.com/questions/34673164/.
[6] 2016. Keras low accuracy classification task. https://stackoverflow.com/questions/38648195/.
[7] 2016. Neural network accuracy optimization. https://stackoverflow.com/questions/39525358/.
[8] 2016. Why can't my CNN learn? https://stackoverflow.com/questions/37229086/.
[9] 2017. How does Keras handle multilabel classification? https://stackoverflow.com/questions/44164749/.
[10] 2017. Keras: Training loss decrases (accuracy increase) while validation loss increases (accuracy decrease). https://stackoverflow.com/questions/47272383/.
[11] 2017. Why does a binary Keras CNN always predict 1? https://stackoverflow.com/questions/45378493/.
[12] 2018. CNN not efficient on my dataset in Keras. https://stackoverflow.com/questions/51749207/.
[13] 2018. CNN train accuracy gets better during training, but test accuracy stays around 40%. https://stackoverflow.com/questions/48594888/.
[14] 2018. CNN with keras, accuracy not improving. https://stackoverflow.com/questions/50079585/.
[15] 2018. Create a square function estimator with Keras. https://stackoverflow.com/questions/48221692/.
[16] 2018. How to improve the performance of CNN Model for a specific Dataset? Getting Low Accuracy on both training and Testing Dataset. https://stackoverflow.com/questions/70554413/.
[17] 2018. Input nodes in Keras NN. https://stackoverflow.com/questions/51930566/.
[18] 2018. Keras overfits on one class Cifar-10. https://stackoverflow.com/questions/51118032/.
[19] 2018. my Keras model does not predict negative values. https://stackoverflow.com/questions/48251943/.
[20] 2018. Non linear Regression: Why isn't the model learning? https://stackoverflow.com/questions/48934338/.
[21] 2018. Simple Keras neural network isn't learning. https://stackoverflow.com/questions/48385830/.
[22] 2019. Accuracy Equals 0 CNN Python Keras). https://stackoverflow.com/questions/58844149/.
[23] 2019. Accuracy Equals 0 CNN Python Keras. https://stackoverflow.com/questions/58844149/.
[24] 2019. Keras CNN intermediate level has no feature changes. https://stackoverflow.com/questions/54923573/.
[25] 2019. Keras CNN model with a wrong ROC curve and low accuracy. https://stackoverflow.com/questions/56914715/.
[26] 2019. Loss doesn't decrease in Pytorch CNN. https://stackoverflow.com/questions/58666904/.
[27] 2019. Low accuracy after training a CNN. https://stackoverflow.com/questions/59325381/.
[28] 2019. Manual predictions of neural net go wrong. https://stackoverflow.com/questions/58609115/.
[29] 2019. My cnn accuracy goes down after adding one more feature. https://stackoverflow.com/questions/55343875/.
[30] 2019. Sudden 50% accuracy drop while training convolutional NN. https://stackoverflow.com/questions/55198221/.
[31] 2019. Super low accuracy for neural network model. https://stackoverflow.com/questions/59278771/.
[32] 2019. tf.keras loss becomes NaN. https://stackoverflow.com/questions/55328966/.
[33] 2020. Getting pretty bad accuracy using CNN Model in Keras. https://stackoverflow.com/questions/65275387/.
[34] 2020. Keras model not training layers, validation accuracy always 0.5. https://stackoverflow.com/questions/60261103/.
[35] 2020. normalize training data with channel means and standard deviation in CNN model. https://stackoverflow.com/questions/63027146/.
[36] 2020. Poor accuracy of CNN model with Keras. https://stackoverflow.com/questions/64522751/.
[37] 2020. Pytorch CNN loss is not changing,. https://stackoverflow.com/questions/60003876/.
[38] 2020. why is my model performing poorly for a keras sequential model? https://stackoverflow.com/questions/64188884/.
[39] 2021. Pytorch CNN not learning. https://stackoverflow.com/questions/65659888/.
[40] 2021. Why does the loss decreases and the accuracy dosen't increases? PyTorch. https://stackoverflow.com/questions/70428592/.
[41] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing neural network architectures using reinforcement learning. In *ICLR'17: International Conference on Learning Representations*.
[42] Houssem Ben Braiek and Foutse Khomh. 2023. Testing Feedforward Neural Networks Training Programs. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 105 (may 2023), 61 pages. https://doi.org/10.1145/3529318
[43] Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *In Neural networks: Tricks of the trade*. Springer, Berlin, Heidelberg, 437–478.
[44] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).
[45] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369.

[46] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs. In *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[47] Nwankpa Chigozie, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation functions: Comparison of trends in practice and research for deep learning. In *arXiv preprint arXiv:1811.03378*.

[48] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Computer Vision and Pattern Recognition (CVPR)*.

[49] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 171–191.

[50] Martín Abadi et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation*. 265–283.

[51] Francois Chollet. 2015. Keras: the Python Deep Learning library. https://keras.io/.

[52] Francois Chollet. 2015. Keras: the Python Deep Learning library. https://keras.io/api/losses/.

[53] Ali Ghanbari, Deepak-George Thomas, Muhammad Arbab Arshad, and Hridesh Rajan. 2023. Mutation-based Fault Localization of Deep Neural Networks. In *ASE'2023: 38th IEEE/ACM International Conference on Automated Software Engineering* (Kirchberg, Luxembourg).

[54] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 108–124.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR'16: IEEE conference on computer vision and pattern recognition*. 770–778.

[56] Robert Hecht-Nielsen. 1992. Theory of the backpropagation neural network. *Neural networks for perception* (1992), 65–93.

[57] Larochelle Hugo, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. 2009. Exploring strategies for training deep neural networks. In *Journal of machine learning research*, Vol. 10.

[58] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *ICSE'20: The ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.

[59] Giancarlo Iannizzotto, Lucia Lo Bello, Andrea Nucita, and Giorgio Mario Grasso. 2018. A vision and speech enabled, customizable, virtual assistant for smart environments. In *2018 11th International Conference on Human System Interaction (HSI)*. 50–56.

[60] Pier Paolo Ippolito. 2022. Hyperparameter Tuning: The Art of Fine-Tuning Machine and Deep Learning Models to Improve Metric Results. In *Applied data science in tourism: Interdisciplinary approaches, methodologies, and applications*. Springer, 231–251.

[61] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2019)*.

[62] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *ICSE'20: The 42nd International Conference on Software Engineering* (Seoul, South Korea).

[63] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 1946–1956.

[64] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR'16: International Conference on Learning Representations*.

[65] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[66] Alex Krizhevsky and Geoff Hinton. 2010. Convolutional deep belief networks on cifar-10. *Unpublished manuscript 40* 7 (2010), 1–9.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional networks. In *NIPS'12: The 25th International Conference on Neural Information Processing Systems*, Vol. 1. 1097–1105.

[68] Yann A. LeCunand Léon, BottouGenevieve B.Orr, and Klaus-Robert Müller. 2012. Efficient backprop. In *Springer*. Berlin, Heidelberg.

[69] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[70] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of IEEE*, Vol. 11. 2278–2324.

[71] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[72] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2024. Replication Package of Paper Titled: "Leveraging Data Characteristics for Bug Localization in Deep Learning Programs". https://doi.org/10.5281/zenodo.14292112 . The replication package and results are available in this GitHub repository: https://github.com/Ruchira-1/Theia.

[73] Sharir Micha and Amir Pnueli. 1978. Two approaches to interprocedural data flow analysis. In *New York University. Courant Institute of Mathematical Sciences*.

[74] Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming Jiang. 2023. Bugs in machine learning-based systems: a faultload benchmark. In *Empirical Software Engineering*, Vol. 28.
[75] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML'10: International Conference on Machine Learning*.
[76] Amin Nikanjam, Braiek Ben Houssem, Morovati Mehdi Mohammad, and Khomh Foutse. 2021. Automatic fault detection for deep learning programs using graph transformations. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 31. 1–27.
[77] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).
[78] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2016. PyTorch: open source machine learning framework. https://pytorch.org/.
[79] Csaky Richard. 2019. Deep learning based chatbot models. In *arXiv preprint arXiv:1908.08835*.
[80] Abhimanyu Roy, Jingyi Sun, Robert Mahoney, Loreto Alonzi, Stephen Adams, and Peter Beling. 2018. Deep learning detecting fraud in credit card transactions. In *2018 systems and information engineering design symposium*. 129–134.
[81] Eldon Schoop, Forrest Huang, and Björn Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *Proceedings of the 2021 CHI Conference Extended Abstracts on Human Factors in Computing Systems*.
[82] Ioffe Sergey and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. 448–456.
[83] Olin Shivers. 1991. DataFlow Analysis and TypeRecovery in Scheme. In *Topics in Advanced Language Implementation: MIT Press*.
[84] Wei Shiyi and Barbara G. Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*.
[85] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[86] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. In *The journal of machine learning research*, Vol. 15. 1929–1958.
[87] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *ICSE'22: The 44th International Conference on Software Engineering*.
[88] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. DeepLocalize: fault localization for deep neural networks. In *ICSE'21: The 43nd International Conference on Software Engineering*.
[89] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *ICSE'21: The 43rd International Conference on Software Engineering*. 359–371.
[90] Yuhao Zhang, Chen Yifan, Cheung Shing-Chi, Xiong Yingfei, and Zhang Lu. 2018. An empirical study on TensorFlow program bugs. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.