DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs

Mohammad Wardat
wardat@iastate.edu
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, USA

Breno Dantas Cruz
bdantasc@iastate.edu
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, USA

Wei Le
weile@iastate.edu
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, USA

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

ABSTRACT

Deep Neural Networks (DNNs) are used in a wide variety of applications. However, as in any software application, DNN-based apps are afflicted with bugs. Previous work observed that DNN bug fix patterns are different from traditional bug fix patterns. Furthermore, those buggy models are non-trivial to diagnose and fix due to inexplicit errors with several options to fix them. To support developers in locating and fixing bugs, we propose DeepDiagnosis, a novel debugging approach that localizes the faults, reports error symptoms and suggests fixes for DNN programs. In the first phase, our technique monitors a training model, periodically checking for eight types of error conditions. Then, in case of problems, it reports messages containing sufficient information to perform actionable repairs to the model. In the evaluation, we thoroughly examine 444 models – 53 real-world from GitHub and *Stack Overflow*, and 391 curated by AUTOTRAINER. DeepDiagnosis provides superior accuracy when compared to UMLUAT and DeepLocalize. Our technique is faster than AUTOTRAINER for fault localization. The results show that our approach can support additional types of models, while state-of-the-art was only able to handle classification ones. Our technique was able to report bugs that do not manifest as numerical errors during training. Also, it can provide actionable insights for fix whereas DeepLocalize can only report faults that lead to numerical errors during training. DeepDiagnosis manifests the best capabilities of fault detection, bug localization, and symptoms identification when compared to other approaches.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

deep neural networks, fault location, debugging, program analysis, deep learning bugs

ACM Reference Format:
Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: Automatically Diagnosing Faults and Recommending Actionable Fixes in Deep Learning Programs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510071

1 INTRODUCTION

Deep Neural Networks (DNNs) are becoming increasingly popular due to their successful applications in many areas, such as healthcare [27, 34], transportation [43], and entertainment [21]. But, the intrinsic complexity of deep learning apps requires that developers build DNNs within their software systems to facilitate integration and development with other applications. The construction of such systems requires popular Deep Learning libraries [18, 32].

Despite the increasing popularity and many successes for using Deep Learning libraries and frameworks, DNN applications still suffer from reliability issues [25, 26, 47]. These faults are harder to detect and debug when compared to traditional software systems, as the bugs are often obfuscated within the DNNs. Therefore, it is important and necessary to diagnose their faults, and provide actionable fixes. To that end, software engineering research has recently focused on improving the reliability of DNN-based software. For instance, there have been studies on characterizing DNN bugs [25, 26, 47], on testing frameworks for deep learning [42], on debugging deep learning using differential analysis [30], and fixing DNNs [15, 46, 48]. There are also frameworks and tools for inspecting and detecting unexpected behavior in DNNs. However, they require that specialists verify the visualization, which is only available upon completing the training phase [3–5, 31, 39].

Due to the complexity of using existing frameworks to debug and localize faults in deep learning software, recent SE research has introduced techniques for automatically localizing bugs [44, 49]. DeepLocalize performs dynamic analysis during training to localize bugs by monitoring values produced at the intermediate nodes of the DNNs [44]. If there is a numerical error, then this approach traces that back to the faulty layer. DEBAR [49] is a static analysis tool that detects numerical errors in the DNNs. While both approaches have significantly advanced the state of the art in

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510071

debugging DNNs, they do not detect bugs that manifest as trends of values (e.g. vanishing gradient, exploding gradient, accuracy not increasing) and do not offer possible fixes.

We propose DeepDiagnosis (DD), an approach for localizing faults, reporting error symptoms, diagnosing problems, and providing suggestions to fix structural bugs in DNNs. Our approach introduces three new symptoms of structural bugs and defines new rules to map fault location to its root cause in DNN programs. We implemented DD as a dynamic analysis tool and compared and contrasted it against state-of-the-art approaches. DD outperforms UMLAUT [38] and DeepLocalize [44] in terms of efficiency and AUTOTRAINER in terms of performance [48]. For example, assume the *unchanged weight symptom*, which occurs when the weights in the network are not changing for several iterations. In that case, DD would identify the root cause as that the *learning rate is too low* or that *the optimizer is incorrect* and then recommend a fix.

In summary, this paper makes the following contributions:

- We study different types of symptoms and propose a dynamic analysis for detecting errors and recommending fixes.
- We introduced DeepDiagnosis (DD) the reference implementation of our approach.
- We evaluated DD against SoTA. We found that DD is more efficient than UMLAUT [38] and DeepLocalize [44]. Also, DD has better performance than AUTOTRAINER [48].
- We provide a set of 444 models that practitioners can use to evaluate their fault localization approaches.
- We make DD available, its source code, evaluation results, and the problem solutions for 444 buggy models at [6].

To the best of our knowledge, DeepDiagnosis provides the fixed location and the concrete fix at the DNN source code level. Our approach detects problems during the training process, and can handle a broad class of problems, e.g., compared to DeepLocalize [44], that do not manifest themselves as numerical errors. It is challenging to provide a correct fix for an observed symptom. Islam *et al.* [26] show that solving a single problem may lead to additional ones. DeepDiagnosis addresses the issue by building the connection between symptoms to root causes. To obtain suitable solutions, we propose a *Decision Tree* to map symptoms to fix.

The rest of the paper is organized as follows: §2 describes the motivation of our approach. §3 describes our dynamic failure symptoms detection algorithm. §4 describes the evaluation of our approach compared with prior works. §5 discusses the threats to validity. §6 discusses related works, and §7 concludes and discusses future work.

2 A MOTIVATING EXAMPLE

In this section, we motivate our work by providing an example to illustrate the complexity of localizing faults and reporting their symptoms in DNN programs.

```
1  model = Sequential()
2  model.add(Dense(128, 50))
3  model.add(Activation('relu'))
4  model.add(Dropout(0.2))
5  model.add(Dense(50, 50))
6  model.add(Activation('relu'))
7  model.add(Dropout(0.2))
8  model.add(Dense(50, 1))
9  model.add(Activation('softmax'))
10 model.compile(loss='binary_crossentropy', optimizer=RMSprop())
11 model.fit(X,Y,batch_size,epoch,validation_data=(X_test,Y_test))
```

Listing 1: Bad Result for Simple Model [2]

Consider the code snippet in Listing 1 from *Stack Overflow* [2] with an example of a DNN. This model showed erratic behavior during training and returns bad results. At line 1, the developer constructed a sequential model and added a dense input layer at line 2 with the activation functions `relu` specified at line 3. Then the developer added a dropout layer at lines 4 and 7. Lines 5 and 8 are dense hidden layers with the activation functions `relu` and `softmax` specified at lines 6 and 9, respectively. The developer then compiled the model at line 10 and trained it using the `fit()` function at line 11. When compiling, the developer must specify additional properties, such as loss function and optimizer. In this example, the developer used as loss `binary_crossentropy` and optimizer `RMSprop()` at line 10. Finally, at line 11, the developer specifies the training data, `batch_size`, epoch, and `validation_data`.

The developer noticed that the DNN program was providing bad accuracy and could not diagnose the problem nor fix it (*Stack Overflow* post [2]) while following the Keras MNIST example [18].

The main issue with the code in Listing 1 is that it handles a *binary classification* problem, and therefore it should **not** use the activation function `softmax` in line 9. As the `softmax` works for *multi-class classifications* problems. Instead, it should use `sigmoid`, as it is the best suited for binary classification and will provide the best accuracy for the task.

Table 1: Result from Motivating Example

Approach	Ouput
UMLAUT	No Output
DeepLocalize	layer 7: Numerical Error in delta Weights
AUTOTRAINER	solution,times,issue_list,train_result,describe 1. selu,0,['relu'],0.5,Using 'SeLU' activation in each layers' 2. bn,0,['relu'],0.5,Using 'BatchNormalization' ... Unsolved.. For more details [6]
DeepDiagnosis	Layer 7: Numerical Error in delta Weights Change the activation function at layer: 8

The current state-of-the-art for DNN fault localization is limited in terms of speed, accuracy, and efficiency. Table 1 summarizes the analysis results from three tools (DeepLocalize [44], UMLUAT [38], AUTOTRAINER [48]) and our approach DeepDiagnosis to diagnose the DNN model in Listing 1. To apply UMLUAT for the above example, we made semantic changes that were validated by the authors [38]. After 104.65 seconds, the training was terminated, with UMLAUT not reporting any problems. To apply DeepLocalize, we followed the instructions in the GitHub repository [7]. DeepLocalize prints the following message after 2.14 seconds: "Layer 7: Numerical Error in delta Weights." This message indicates that there is a numerical error in the backpropagation stage during training. Indicating fault location, but it does not help developers to fix the problem. To apply AUTOTRAINER, we followed the instructions in the GitHub repository [8]. After performing the training phase, AUTOTRAINER did not solve the problem and took 495.83 seconds. Specifically, AUTOTRAINER detects a Dying ReLU symptom, but it does not provide the fault location – whether it is in line 3 or 6. AUTOTRAINER tries to automatically fix the issue by trying different

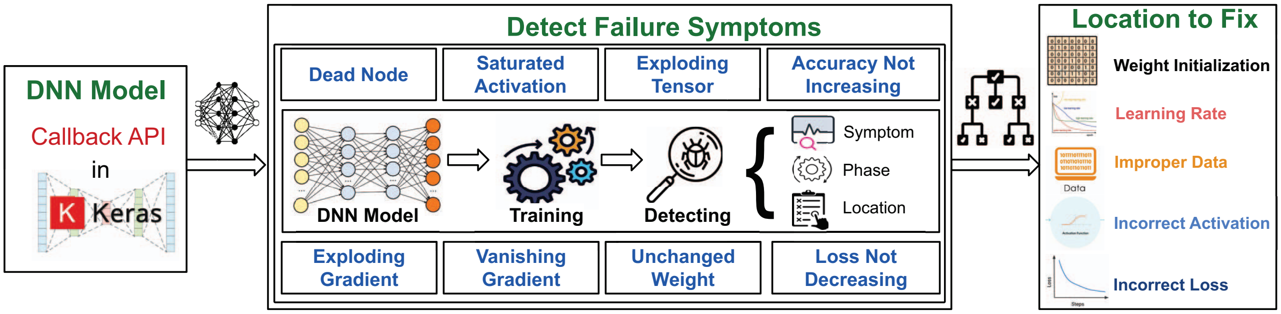

Figure 1: Overview of DeepDiagnosis.

strategies (i.e., substituting activation functions, adding batch normalization layer, and substituting initializer), which, unfortunately, are unsuccessful.

Our approach DeepDiagnosis correctly reports the fault location and its symptoms after 35.03 seconds. Also, it provides a suggestion to perform a fix in the form of a message. Specifically, DeepDiagnosis reports that the bug is located in the backpropagation stage of layer 7 at line 8. Also, it prints out a numerical message: "Error in delta Weights", which indicates the type of the symptom. It also reports that the root cause is the activation function in layer 8 at line 9 (`softmax`). Finally, it answers the developer's question – there is indeed a problem with the activation function in the last layer and not in the training dataset.

3 APPROACH

In this section, we provide an overview of our approach for fault localization. We provide descriptions of failure symptoms and their root causes. Also, we describe the process of mapping symptoms to their root causes.

Our approach monitors the key values during training, like weights and gradients. During training, it analyzes the recorded value to detect symptoms and determine whether a training problem exists. If a symptom is detected, our approach invokes a Decision Tree to diagnose/repair information based on a set of predetermined rules. Otherwise, the training will terminate with the trained model and report the model is correct.

3.1 An Overview

Figure 1 shows an overview of our approach for fault localization, DeepDiagnosis, and for suggesting locations fix. DD starts by receiving as input the initial model architecture with a training dataset and passing our callback method as a parameter to the `fit()` method (Figure 1 left component). Keras callbacks are a set of methods that enable developers to check their model's intermediate features (e.g., weights, gradients). Also, callbacks enable the developers to inspect the model's behavior during training. Our callback approach is inspired by prior work [14, 44]. In particular, our callbacks allow capturing and recording the key values (i.e., weight, gradient, etc.) during feed-forward and backward propagation stages (Figure 1 middle component). Then DD applies a dynamic detector during training to report different symptoms at different stages based on error conditions (see Section 3.2 for more details). If DD detects a symptom, it further analyzes the recorded key values to determine the input model's probable location for the fix (Figure 1 right component). Finally, DD reports the symptom type, which layers and stage the symptom was detected, and suggests a location fix.

3.2 Failure symptoms and root causes

Our goal is to detect failure symptoms as soon as possible during development. So that if the model is incorrect, developers would not have to wait until the end of the training to find that model has low accuracy, thus wasting computational resources. To that end, we collected 8 types of failure symptoms and their root causes from previous work in the AI research community [23, 24, 33, 40]. We provide more details of each of the symptoms and their root causes below.

3.2.1 *Symptom #1 Dead Node.*

The Dead Node symptom takes place when most of a neural network is inactive. For example, assume that most of the neurons of a DNN are using the ReLU activation function, which returns zero when receiving any negative input. If the majority of the neurons receive negative values (e.g., due to a high learning rate), they would become inactive and incapable of discriminating the input. The DNN would end up with poor performance [48]. To identify this symptom, we compute the percentage of inactive neurons per layer. If the majority of the neural network is inactive, then we call it Dead Node.

Root Causes: This problem is likely to occur when [16]: (1) learning rate is too high/low. (2) there is a large negative bias. (3) improper weight or bias initialization.

3.2.2 *Symptom #2 Saturated Activation.*

The Saturated Activation symptom takes place when the input to the logistic activation function (e.g., *tanh* or *sigmoid*) reached either a very large or a very small value [23]. At the saturated point, the function results would equal zero or be close to zero, thus leading to no weight updates. Our experiments show [6] that the behavior of *sigmoid* and *tanh* have a minimum saturated point at x=-5 and a maximum saturated point at x=5. Previous work showed that the saturated function affects the network's performance and makes the network difficult to train [23, 45].

Root Causes: This problem is likely to occur when [19]: (1) the input data are too large or too small; (2) improper weight or bias initialization; (3) learning rate is too high or too small.

3.2.3 *Symptom #3 Exploding Tensor:*

The Exploding Tensor symptom takes place when the tensors' values become too large, leading to numerical errors in a feed-forward stage. For example, if

Table 2: Methods for Detecting Failure Symptoms

ID	Method Name	Input	Output	Description
S1	ExplodingTensor ()	Weight, Δ Weight, and Layer Output	T | F	The procedure detects any numerical error such as infinite, NaN (Not a Number), or zero. To that end, it computes the input's mean value. Then, it checks for a numerical error is detected. In case of error, it returns **True**, otherwise **False**.
S2	UnchangeWeight ()	Weight, Δ Weight, Layer Output	T | F	The procedure stores the value for a given number of steps (N = 5). Then it compares the value for the current step with the mean value stored in for previous (N = 5) steps. The evaluation takes place for every given number of steps. The procedure returns **True** if the value is not changing, otherwise **False**.
S3	SaturatedActivation ()	Input of Activation Function	T | F	The procedure detects if the tanh or sigmoid activation functions, or other logistic functions are becoming saturated. It does so by checking if their input has reached either a maximum or minimum value. Saturated functions' derivatives would be equal to zero at those points. The procedure counts the activity of a close or greater node than to the (Max_Threshold = 5) or less than (Min_Threshold = -5) of the activation function; If the percentage of total activity nodes is greater than the (Threshold_Layer = 0.5) percent of the nodes are saturated, the procedure returns **True**, otherwise **False**.
S4	DeadNode ()	Relu Output	T | F	This procedure takes the output of Rectified Linear Unit (ReLU) activation function as input, then computes how many inactive nodes dropped below (Threshold = 0.0). If the percentage of inactive nodes is greater than (Layer_Threshold = 0.7) it returns **True**, otherwise **False**.
S5	OutofRange ()	Output of last layer	T | F	The procedure detects if the activation function's output is becoming out of range for the labeling training dataset Y. To that end, it finds the range (maximum and minimum) of the activation function's output. Then compare it with Y labeling data. If the value is out of the boundary, the procedure returns **True**, otherwise **False**.
S6	LossNotDecreasing ()	Loss Value	T | F	The procedure stores the loss value for every number of steps (N = 5), then compares the loss value for the current step with the mean value of losses stored in the previous (N = 5) steps. The evaluation happens for every number of steps (N = 5). The procedure returns **True** if the loss is not decreasing, otherwise **False**.
S7	AccuracyNotIncreasing ()	Accuracy Value	T | F	The procedure stores the accuracy value for every number of steps (N = 5), then compares the accuracy value for the current step with the mean value of accuracy stored in previous (N = 5) steps. The evaluation happened every number of steps (N = 5). The procedure returns **True** if accuracy is not increasing, otherwise **False**.
S8	VanishingGradient ()	Delta Weight	T | F	This procedure detects the Vanishing Gradient problem by checking the gradients when they become extremely small or drop to zero. The procedure computes the mean of the gradients' absolute values, then checks if their means drop below a specified (Threshold = 0.0000001). In the case of a positive detection, it returns **True**, otherwise **False**.

This table shows procedures descriptions from [1, 10, 20, 37, 48]. T|F indicates that the procedure returns True| False respectively.

Table 3: Methods for Mapping from Failure Symptoms to Location Fix

No	Method Name	Input	Output	Description
C1	ImproperData ()	Training Data	T | F	Check if the maximum and minimum value of training dataset lie within specific range of [-1, 1]. If the value within the boundary, the procedure returns True. Otherwise, False.
C2	WeightInitialization ()	Weight for each layer	T | F	This procedure checks the variance of weight inputs across layers to determine if a neural network has been poorly initialized. The procedure checks if the variance of weights per layer is equal or very close to 0 (Min_Threshold = 0.00001), or if it exceeds the (Min_Threshold = 10), the procedure returns **True**. Otherwise, **False**.
C3	TuneLearn ()	Learning rate, Weight, and Δ Weight	L | H	The procedure evaluates the learning rate heuristically by computing the ratio of the norm of the gradient weight to the norm of weight for each layer. This ratio should be somewhere around (Learn_Threshold = 1e-3). If it is lower than (Learn_Threshold = 1e-3), then the learning rate might be too **Low**. If it is higher than (Learn_Threshold = 1e-3), the learning rate is likely too **High**.

This table is showing all the functionality of the procedures. T|F indicates the procedure return True| False respectively. L|H indicates the procedure return Low| High respectively. We borrowed these methods from existing literature [1, 10, 20, 37, 48]

the weight or output layer grows exponentially more than expected, becoming either infinite or *NaN* (not a number). Eventually, this problem causes a numerical error, making it impossible for the model to learn.

Root Causes: This problem is likely to occur when [20, 28]: (1) the learning rate is too large; (2) there exist improper weight or bias initialization, or improper input data.

3.2.4 *Symptom #4 Accuracy Not Increasing & Symptom #5 Loss Not Decreasing*.

Both symptoms Accuracy Not Increasing and Loss Not Decreasing are very similar. The Accuracy Not Increasing symptom takes place when the accuracy of a target model is not increasing for N steps, but instead, it is decreasing or fluctuating during training. While for the Loss Not Decreasing symptom, the loss metric is the one that is not decreasing for N steps but is fluctuating. These behaviors indicate that the network will not achieve high performance. These symptoms are often caused by the incorrect selection of DNN hyperparameters [37], such as loss function, activation function for the last layer, learning rate, optimizer, or batch size.

Root Causes: This problem is likely to occur when [20, 28]: (1) there exist improper training data; (2) the number of layers is too large/small; and (3) the learning rate is very high/low; and (4) there exist incorrect activation functions.

3.2.5 *Symptom #6 Unchanged Weight*.

The Unchanged Weight symptom takes place when the DNN weights do not have a noticeable influence on the output layers. This behavior leads to unchanging parameters and network stacks, which further prevents the model from learning [19, 44].

Root Causes: This problem is likely to occur when [19, 44]: (1) learning rate is very low; (2) the optimizer is incorrect; (3) there exist incorrect weights initialization; and (4) there exists incorrect loss/activation at the last layer.

3.2.6 *Symptom #7 Exploding Gradient*.

This problem occurs during the back-propagation stage. In it, gradients are growing exponentially from the last layer to the input layer, which leads to non-finite values, either infinite or *NaN* (not a number). This issue makes learning unstable and sometimes even impossible. Consequently, updating the weights becomes very hard, and the training model ends up with a high loss or very low accuracy values.

Root Causes: This problem is likely to occur when [20, 28]: (1) the learning rate is very high; (2) there is an improper weight or bias initialization; (3) there are improper data input; and (4) the batch size is very large.

3.2.7 ***Symptom #8 Vanishing Gradient.*** The Vanishing Gradient problem occurs during the backward stage. When computing the gradient of the loss concerning weights using partial derivatives, the value of the gradient turns out to be so small or drops to zero. The problem causes major difficulty if it reaches the input layer, which will prevent the weight from changing its value during training. Since the gradients control how much the network learns during training, the neural network will end up without contributing to the prediction task or leading to poor performance [41, 48].

Root Causes: This problem is likely to occur when [29]: (1) the network has too many layers; (2) the learning rate is low; (3) the hidden layers improperly used *Tanh* or *Sigmoid*; and (4) there exists the incorrect weight initialization problem.

3.3 Detecting Failure Symptoms

In Table 2 from Method S1 to S8, we describe the failure symptoms discussed in Section 3.2, using its name, input/output, and the detection procedure. Algorithm 1 shows an example of a dynamic analysis procedure, which DeepDiagnosis uses to detect failure symptoms during training (Table 2 *Description*). Also, the Algorithm 1 reports failure locations, such as in which layer and phases (i.e., feed-forward and backward propagation). In case a failure is detected, the algorithm will trigger the *Mapping()* procedure to identify the location in the original DNN source code. By doing so, it will localize the bug and determine the optimal fix.

At line 1, Algorithm 1 iterates over the training epochs, with the training dataset divided into batches. Line 3 shows the division of the training dataset into a mini-batch. On lines 2-28, the algorithm runs one batch of the training dataset before updating the internal model parameters. The neural network can be divided into two stages: First, the forward stage, in which the algorithm performs dynamic analysis and symptom detection, including Numerical Error, Dead node, Saturated Activation, and Out of Range, at lines 4-12. Second, the backward stage, in which the algorithm performs dynamic analysis to detect additional symptoms, such as Numerical Error, Vanishing Gradient, and Unchanged weight at lines 23-28.

3.3.1 Feed-forward stage. At lines 5 & 6 of the Algorithm 1, it computes the output of a feed-forward before and after applying the activation function. At line 7, it invokes the *ExplodingTensor()* procedure (S1 in Table 2) to determine if the output contains a numerical error obtained from the output value before/after applying activation function, respectively. If there is an error, the algorithm reports the NS message as shown in Table 5. Next, it invokes the Mapping() procedure from the decision tree in Figure 2 by providing the symptom (NS), location, stage (FW), and layer (L). The decision tree returns the best actionable fix for the model (see Section 3.4 for more details).

At line 8, the Algorithm 1 invokes the *UnchangeWeight()* (S2 in Table 2) procedure to detect whether the output before/ after applying the activation function is no longer changing across steps. If the procedure indicates that the value does not change for N iterations, we follow [44] and set N=5. The *UnchangeWeight()* procedure can be applied either to the output before/after the activation function. The algorithm reports the message UCS, as shown in Table 5. At line 9, the Algorithm invokes the SaturatedActivation () procedure (S3 in Table 2) for the layer that has a logistic activation function (i.e., `tanh` or `sigmoid`) to determine if the layer is becoming saturated. This procedure takes two arguments, the value before applying the activation function (V_1) and the name of the activation function (V_2.name). If the procedure determines that the layer is saturated, the algorithm reports the message SAS as shown in Table 5.

At line 10, the Algorithm 1 invokes the *DeadNode()* procedure (S4 in Table 2) to check the layers that use the Rectified Linear Unit (ReLU) activation function. The goal is to determine if the output after applying the activation function has dropped below a threshold [48]. This procedure is invoked only after applying the activation function. The algorithm reports the message DNS as shown in Table 5 when the error is detected. Similarly, at line 11, it invokes the *OutofRange()* procedure (S5 in Table 2) in the last layer. The goal is to determine if the developer has chosen the correct activation function. The algorithm reports the message ORS as shown in Table 5 if the error is detected.

In lines 13 & 15 the algorithm interprets and validates how well the model is doing by computing the loss and accuracy metrics, respectively. Then it determines if there is any numerical error in those metrics at lines 14 & 16, respectively. The algorithm invokes *LossNotDecreasing()* and *AccuracyNotIncreasing()* (S6 & S7 in Table 2) to check if the loss or the accuracy has not changed for a long time. In both cases, the algorithm reports a message LNDS or ANIS as shown in Table 5.

3.3.2 Back propagation stage. During this stage, the Algorithm 1 computes the gradient of loss function Δ Weight for the weight by chain rules in each iteration. At line 24, the algorithm invokes *Backward()* to apply stochastic gradient descent, and this function returns the Weight and Δ Weight in each iteration. At line 25, the algorithm invokes the *VanishingGradient()* procedure (S8 in Table 2) and passes Δ Weight to check if the gradients become extremely small or close to being zero. In the same way, at line 26, the algorithm can determine if there is a numerical error in the Weight or the gradient weight in each layer by invoking the *ExplodingTensor()* procedure (S1 in Table 2). The backpropagation algorithm works if the Weight is updated using the gradient method and the loss value keeps reducing, to check if the backpropagation works effectively. In the backward propagation, we also invoke the *UnchangeWeight()* procedure (S2 in Table 2) to detect whether the weight or Δ Weight is no longer changing across steps. If any procedure decides that there is an issue, then the algorithm will return a message to indicate the type of symptom as shown in Table 5, L represents a faulty layer number. Then the algorithm invokes *Mapping()* and passes the symptom, location, and layer to find the best actionable change to fix the issue in the model. Finally, if the algorithm did not detect any type of symptom, it will terminate after finishing the training at line 29 and print a message indicating that there is no issue in the model (CM).

Table 4: Abbreviation of Actionable Changes

No	Message Guideline	Abbreviation
1	Improper Data	MSG0
2	Change the loss function	MSG1
3	Change the activation function	MSG2
4	Change the learning rate	MSG3
5	Change the initialization of weight	MSG4
6	Change the layer number	MSG5
7	Change the optimizer	MSG6

Table 5: Abbreviation of Failure Symptoms

No	Symptoms	Abbreviation
1	Numerical Errors	NS
2	Unchanged weight	UCS
3	Saturated Activation	SAS
4	Dead Node	DNS
5	Out of Range	ORS
6	Loss Not Decreasing	LNDS
7	Accuracy Not Increasing	ANIS
8	Vanishing Gradient	VGS
9	Invalid Loss	ILS
10	Invalid Accuracy	IAS
11	Correct Model	CM

3.4 Mapping Symptoms to Location fix

Decision Tree: The main goal of this step is to mitigate manual effort and reduce the time for diagnosing and fixing bugs. To that end, the *Mapping()* procedure in Algorithm 1 provides fix suggestions based on the detected failure symptoms. Figure 2 shows a representation of the Decision Tree which the *Mapping()* procedure uses to provide a fix recommendation.

The Decision Tree consists of 24 rules, which corresponds to decision paths. Each rule provides a mapping from failure symptoms and detected locations to actionable changes. The tree defines a binary classification rule which maps instances in the format problem (Symptom, Location, Layer) into one of seven classes of changes (Table 4). In the decision tree, the root node represents the problem, orange nodes the symptoms, blue nodes the locations, gray nodes the layer type, green nodes, the conditions, and red nodes the actionable changes. Table 3 shows the methods `Data()`, `Weight()` and `Learn()`, which are used to compute the conditions. Each Decision Tree instance maps a path from the root to one of the leaves.

For example, assume that a developer wants to check the code in Listing 1. To that end, the developers can use the Algorithm 1 to verify the model. The algorithm invokes the *Mapping()* procedure (line 26) by passing the symptom NS, location, stage BW (backward), and layer (7). This procedure traverses the path under the NS node in the Decision Tree (Figure 2). Since the problem occurred in the BW stage, the algorithm takes the right path to satisfy the condition. Then, it verifies the layer type (7). Since it finds an issue in the layer, the procedure returns the message MSG2 – Change the activation function (Table 4).

Algorithm 1: Failure Symptoms Detection

```
input  : Training data (input, label), DNN program
output : Failure symptoms and locations (layers, iterations, epoch)
1  for $e \leftarrow 0$ to $epochs$ do
2      for $i \leftarrow 0$ to $Length(input)$ Step $batchsize$ do
3          $X \leftarrow input[i]$; $Y \leftarrow label[i]$
4          for $L \leftarrow 0$ to $Length(Layers)$ do
5              $V_1 \leftarrow Layer[L].Forward(X)$
6              $V_2 = Layer[L].Activation(V_1)$
7              if $ExplodingTensor(V_2|V1)$ then return NS,
                 $Mapping(NS, FW, L)$
8              if $UnchangeWeight(V_2|V_1)$ then return UCS,
                 $Mapping(UCS, FW, L)$
9              if $Saturated(V_1, V_2.name)$ then return SAS,
                 $Mapping(SAS, FW, L)$
10             if $DeadNode(V_2)$ then return DNS,
                 $Mapping(DNS, FW, L)$
11             if $OutofRange(V_2, Y)$ && $L == Last$ then return
                 ORS, $Mapping(ORS, FW, L)$
12             $X \leftarrow V_2$
13         $Loss \leftarrow ComputeLoss(V_2, Y)$
14         if $Loss$ is equal to NaN OR $inf$ then return ILS,
             $Mapping(ILS)$
15         $Accuracy \leftarrow ComputeAccuracy(V_2, Y)$
16         if $Accuracy$ is equal to NaN OR inf OR 0 then
17             return IAS, $Mapping(IAS)$
18         if $LossNotDecreasing(Loss)$ then
19             return LNDS, $Mapping(LNDS)$
20         if $AccuracyNotIncreasing(Accuracy)$ then
21             return ANIS, $Mapping(ANIS)$
22         $dy \leftarrow Y$
23         for $L \leftarrow Length(Layers)$ to 0 do
24             $V_3, W[L] \leftarrow Layer[L].Backward(dy)$
25             if $VanishingGradient(W[L])$ then return VGS,
                 $Mapping(VGS, BW, L)$
26             if $ExplodingTensor(V_3|W[L])$ then return NS,
                 $Mapping(NS, V_3|DW, L)$
27             if $UnchangeWeight(V_3|W[L])$ then return UCS,
                 $Mapping(UCS, V_3|DW, L)$
28             $dy \leftarrow V_3$
29 return CM
```

Heuristics: We developed a set of heuristics based on the root causes (see Section 3.2). There are three main root causes: (1) Data Preparation; (2) Parameter Tuning; and (3) Model Architecture. For Data Preparation, the algorithm checks if the data is normalized (C1 - *ImproperData()* in Table 3). For Parameter Tuning, our approach checks if the hyperparameters (such as learning rate) were assigned correctly. Also, to check if the weights were initialized correctly, the algorithm invokes the *WeightInitialization()*. The *TuneLearn()* procedure verifies whether the learning rate is very high or very low (C2 and C3 in Table 3, respectively). For model architecture, the algorithm searches for a relation between the location and the stage of the symptom. It performs this step to pinpoint which APIs are being misused in the model (e.g., loss, activation function).

We collected the root causes for each symptom from previous work [23, 24, 33, 40] (more details in Section 3.2). To arrive at a possible fix for a given symptom, we choose the most frequent root cause. We follow this approach as our findings show that changes in the order we check for the possible root causes do not affect the results, only on the total time to arrive at a solution. For example, assume that a model has the Dead Node symptom. In terms of frequency, improper data tends to happen more often than weight and learning rate. If the three root causes are correct, our approach checks the model architecture, which is the least common in this case. Thus, arriving at an improper activation function as the root cause of this symptom.

4 EVALUATION

In the evaluation, we answer the following research questions:

- RQ1 (Validation): Can our technique localize the faults and report the symptoms in Deep Learning programs effectively?
- RQ2 (Comparison): How does our technique for fault localization compared to existing methodologies in terms of time and effectiveness?
- RQ3 (Limitation): In which cases do our technique fail to localize the faults and report the symptoms?
- RQ4 (Ablation): To what extent does each type of symptom we developed contribute to the overall performance of DeepDiagnosis?

4.1 Experimental setup

4.1.1 Implementation. We implemented DeepDiagnosis on top of *Keras* 2.2.0 [18] and *TensorFlow* 2.1.0 [32]. Also, we implemented Algorithm 1 by overriding the method called *(on_epoch_end(epoch, logs=None)*. For the Decision Tree in Figure 2, we implemented it as a decision rule consisting of a set of conditional statements. The overridden method invokes the decision tree once upon detecting a symptom. Then it passes the symptom type as a parameter for the decision tree.

We conducted all the experiments on a computer with a 4 GHz Quad-Core Intel Core i7 processor and 32 GB 1867 MHz DDR3 GB of RAM running the 64-bit iMac version 4.11.

4.1.2 Benchmark. In total, we collected 548 models from prior work [7, 38, 48]. From these, we removed 104 RNN models, as our approach does not support them. The resulting 444 models are composed of 53, which are known to **have bugs** from [7, 38]. We refer to these 53 models SGS benchmark as they come from **S**tackOverflow, **G**itHub, and ***S****choop et al.* [38]. Also, the 391 models from [48] are in the compiled *.h5 format. The remaining 391 models are divided into two sets. In particular, the first with 188 correct models – **without** any known bugs – and the second with 203 buggy models – **with** bugs.

Most machine learning developers share the source code or the trained weights of their models in *.h5 format. To allow others to improve the understanding of how a model operates and inspect it with new data, we implemented the *Extractor* tool [6]. It extracts the DNN source code from a *.h5 file. The *Extractor* follows three steps to generate the Keras source code: first, it saves the model's layer information to a JSON file. Then, it generates the Abstract Syntax Tree (AST) from the JSON file. Finally, it converts the AST to the source code.

To build the ground truth for the SGS benchmark, we manually reviewed all models and their respective answers from *Stack Overflow* and commits from GitHub. We perform this verification process to determine the exact bug location and its root causes. For the remaining 391 models - 203 **buggy** models and 188 **not buggy** models, we used our *Extractor* to generate the source code for each model before/after performing a fix; we used the difflib [22] module to generate the diff file from the fixed model. We use the diff to locate the changes in the model, thus locating the root causes and the actual location of its corresponding fix. We consider a model successfully repaired if its accuracy has improved after the fix.

4.1.3 Results Representation. Table 6 shows the summarized evaluation results of the following approaches: UMLUAT [38], DeepLocalize [44], AUTOTRAINER [48], and our approach DeepDiagnosis. Please refer to the reproducibility repository [6] for the complete table. The first column shows the source of the buggy model. The second column lists the model ID. The third column provided the *Stack Overflow* post # and the model name from GitHub repositories, collected by Wardat *et al.* [44], and also the name of the model introduced by Schoop *et al.* [8], respectively. To compare our approach with the results generated from previous approaches, we reported the results in the following columns (from left to right): Time, Identify Bug (IB), Fault Localization (FL), Failure Symptom (FS), and Location Fix (LF), and Ablation (AB). Time, in seconds, reports how long each tool takes to report its results. The columns Identify Bug (IB) and Fault Localization (FL) show whether the approach successfully identifies the bug and the fault location. Failure Symptom (FS) and Location Fix (LF) columns show whether the tool correctly reports a symptom and an actionable change (model repair fix). Finally, the Ablation (AB) column shows which of the procedures listed in Table 2 detects the failure symptoms. Under each approach, the "Yes" and "No" status indicates whether it has successfully reported the target problem. Also, the "—" status denotes if the approach does not yet support the target problem. Lastly, we use method ID in Table 2 to indicate which procedure is used to detect the failure symptom.

Table 7 summarizes the analysis results from four approaches using benchmarks collected from three different sources [38, 44, 48]. The second column (Total) lists the total number of buggy models for each dataset. To compare our approach with previous approaches, we reported Time, in seconds, the average time each tool takes to report its results for each dataset. To mitigate randomness during the training model, we followed the procedure described in [48] and ran each model 5 times. The column Identify Bug (IB) shows how many each approach successfully identifies the bug from each dataset. Our approach is capable of handling eight types of symptoms with different types of datasets using different types of model architectures. Table 8 shows the number of symptoms detected from different types of datasets.

4.2 RQ1 (Validation) and RQ2 (Comparison)

Table 6 and 7 show the evaluation results for RQ1 and RQ2.

DeepDiagnosis (DD) has correctly identified 46 out of 53 buggy models from the SGS benchmark. DD correctly reported the fault location for 34 models and the failure symptoms for 37 models. Also, DD correctly identified the actionable changes for 28 out of 53 faulty models. Lastly, DD identified 138 out of the 203 buggy

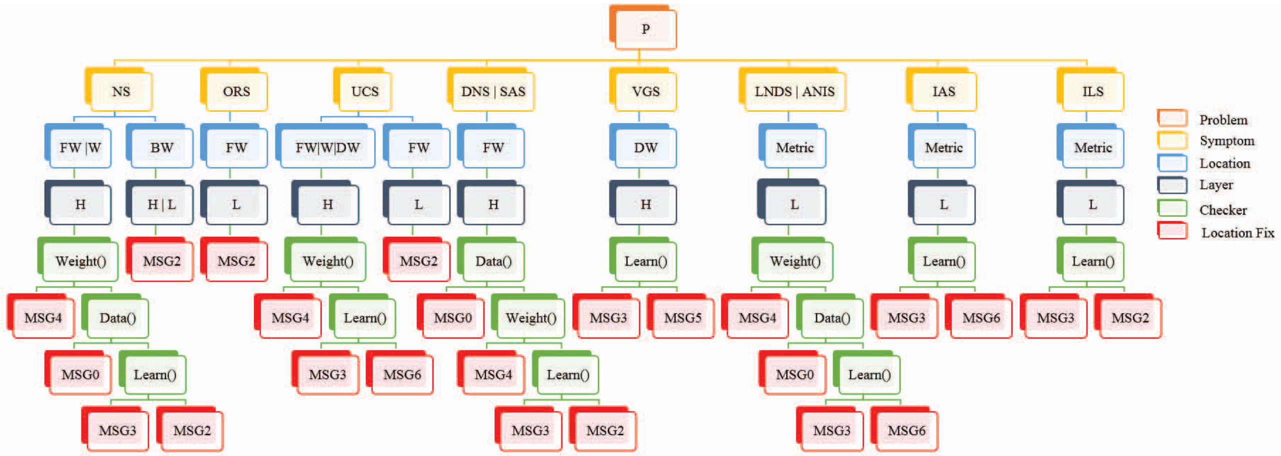

Figure 2: Mapping Symptoms to Fix Location

Table 6: Comparing the Results from UMLAUT (UM), DeepLocalize (DL), AUTOTRAINER (AT) and DeepDiagnosis (DD)

			Time				Identify Bug (IB)				Fault Localization (FL)				Failure Symptom (FS)				Location Fix (LF)				AB
	No	Post #	UM	DL	AT	DD	UM	DL	AT	DD	UM	DL	AT	DD	UM	DL	AT	DD	UM	DL	AT	DD	
Stack Overflow [7]	1	48385830	0.39	2.14	103.91	8.27	Yes	Yes	Yes	Yes	Yes	Yes	—	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	#1
	2	44164749	188.61	111.56	197.90	242.34	No	Yes	No	No	No	Yes	—	No	No	Yes	No	No	No	No	No	No	—
	3	31556268	—	1.2	—	12.48	—	Yes	—	Yes	—	No	—	Yes	—	Yes	—	Yes	—	No	—	Yes	#7
	4	50306988	1.9	3.57	93.60	1.75	No	Yes	Yes	Yes	No	Yes	—	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	#1
	5	48251943	—	706.83	—	1.61	—	No	—	Yes	—	No	—	Yes	—	No	—	Yes	—	No	—	Yes	#5
	6	38648195	5.4	25.92	85.38	15.12	Yes	Yes	No	Yes	No	Yes	-	Yes	No	Yes	No	Yes	No	No	No	Yes	#1
GitHub [7]	7	GH #1	128.67	11.80	6524.21	44.90	Yes	Yes	Yes	Yes	No	No	—	No	No	No	Yes	No	No	No	Yes	No	#1
	8	GH #2	—	8432.06	—	1001.40	—	No	—	No	—	No	—	No	—	No	—	No	—	No	—	No	-
	9	GH #3	—	31.69	—	2.17	—	Yes	—	Yes	—	Yes	—	Yes	—	No	—	Yes	—	No	—	Yes	#5
	10	GH #4	36.58	102.44	315.61	102.96	Yes	Yes	Yes	Yes	No	No	—	No	Yes	No	Yes	No	No	No	Yes	No	#4
	11	GH #5	18.95	164.70	173.92	140.58	Yes	Yes	No	Yes	No	Yes	—	Yes	No	Yes	No	Yes	No	No	No	No	#2
	12	GH #6	—	9568.09	12.57	118.59	—	No	No	No	—	No	—	No	—	No	no	No	—	No	No	No	-
Schoop *et al.* [8]	13	A1 (C-10)	1.77	18.39	43.96	2.75	Yes	Yes	Yes	Yes	Yes	Yes	—	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	#5
	14	A2 (C-10)	1.50	44.93	18.36	10.44	Yes	Yes	No	Yes	No	Yes	—	Yes	Yes	Yes	No	No	Yes	No	No	No	#1
	15	A3 (C-10)	348.88	44.89	119.54	5.03	Yes	Yes	Yes	Yes	No	No	—	No	Yes	No	No	Yes	Yes	No	No	No	#1
	16	B1 (C-10)	347.21	10.65	80.38	2.17	Yes	Yes	Yes	Yes	No	No	—	No	Yes	Yes	Yes	Yes	Yes	No	No	No	#1
	17	B2 (C-10)	3.42	45.02	16.90	5.44	Yes	Yes	No	Yes	No	Yes	—	Yes	Yes	No	No	Yes	Yes	No	No	Yes	#1
	18	B3 (C-10)	1605.99	45.54	15.49	15.49	Yes	Yes	No	No	No	No	—	No	Yes	No	No	No	Yes	No	No	No	—
Total							26	45	24	**46**	3	26	—	**34**	17	23	19	**37**	15	0	8	**28**	—

C-10: indicates to the model using CIFAR-10 dataset, and F-M: indicates to the model using Fashion-MNIST dataset.

Table 7: Runtime Overhead vs. Problem Detects

Dataset	Total	Time				Identify Bug (IB)			
		UM	DL	AT	DD	UM	DL	AT	DD
Stack Overflow **[7]**	**29**	46.16	421.39	771.56	**103.74**	10	27	16	**26**
GitHub [7]	**11**	46.16	2613.6	148.41	**137.82**	4	7	3	**9**
Schoop *et al.* [8]	**12**	193.52	93.17	3491.32	**1020.20**	12	11	5	11
Blob [9]	**48**	—	113.14	112.6	564.19	—	44	48	34
Circle [9]	**71**	—	148.63	84.37	1078.14	—	63	71	47
MNIST [9]	**38**	290.87	16.68	4741.53	1265.02	26	38	38	31
CIFAR-10 [9]	**46**	121.39	22.4	10653.63	3282.83	46	46	46	26

models from the AUTOTRAINER dataset, correctly reporting fault location, failure symptoms, and actionable changes.

DeepLocalize (DL) [44] identified 45 out of the 53 models from the SGS benchmark and indicated fault locations for 26. It reported symptoms for only 23 models, but it cannot provide any suggestions to fix these faults. Regarding the AUTOTRAINER dataset, DL identified 191 out of the 203 buggy models and correctly reported their fault location. However, DL did not provide any suggestions for fixing those models. Lastly, DL can only detect bugs related to numerical errors.

AUTOTRAINER (AT) [48] For the 53 models (SGS benchmark), AT identified 24 buggy models. Out of these, AT successfully reported symptoms for only 19. AT was only able to repair 8 models. DD can handle more varieties of semantically related errors than AT, as shown in Table 6. Please refer to [48] for AT's evaluation results while analyzing its dataset.

UMLUAT (UM) [38] identified 26 buggy models out of the 53 from the SGS benchmark and found the fault locations for 3. Also, UM reported the symptoms for 17 models and provided the location fix for 15 out of 53. UM correctly identified models and reported possible fix solutions to problems from 72 out of 203 buggy models of the AUTOTRAINER dataset. UM only supports classification problems, while DD supports additional types, such as regression and classification.

To evaluate the approaches' overall performance, we collected their total execution time while analyzing the benchmarks. Figure 3 shows the results. UM, DL, AT, and DD require on average 46.16, 421.39, 771.56, and 103.74 seconds, respectively, for all the

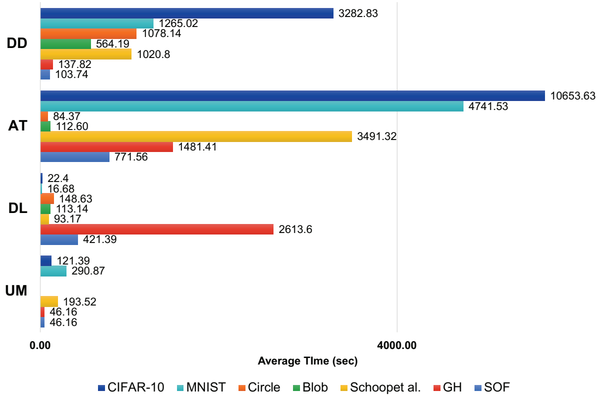

Figure 3: Comparison between UMLUAT (UM) VS DeepLocalize (DL) VS AUTOTRAINER (AT) VS DeepDiagnosis (DD) in terms of seconds

Stack Overflow (SOF) benchmarks. For the GitHub (GH) benchmark, the four approaches require on average 46.16, 2613.60, 148.41, and 137.78 seconds, respectively. For the Schoop *et al.*’s [38] benchmark, the four approaches take on average 193.52, 93.17, 3491.32, and 1020.80 seconds, respectively. For the AUTOTRAINER dataset, the four approaches require, on average, 4159.25, 4157.36, 170156.70, and 74408.07 seconds, respectively, to complete their analysis. Lastly, the overall average time for UM, DL, AT, and DD, for all benchmarks is 2972.23, 8388.21, 106490.05, and 44914.17 seconds, respectively.

DD’s runtime overhead is mainly due to its online dynamic analysis. DD runs its dynamic analysis on the internal parameters of the neural networks, such as the changes of weights and gradients, during the training phase. DD is the most efficient for *Stack Overflow* and Schoop *et al.*’s model and is slower than UM on the GitHub models. The reason is that DD collects more information than UM during training and checks additional types of error conditions.

DD is faster than AT on all benchmarks except for the Blob and Circle datasets. That is because AT checks the target model after finishing the training phase. DD requires additional time because it validates the model at the end of each epoch during training, and the number of epochs for these models is between 200 to 500.

4.3 RQ3 (Limitation)

Out of 52 programs, our technique failed to identify faults in 6 and localize faults in 18. DD failed to report symptoms for 15 programs and failed to provide the location of fix for 24 (Tables 2 and 6). In the following, we provide a few examples of failed fault localization.

Our technique does not yet support model with `fit_generator()` instead of `fit()` function. `fit_generator()` is used for processing a large training dataset that is unable to load into the memory [17]. In the future, we plan to cover more APIs (such as `fit_generator()`).

Both #47 (B3 (C10)), and #53 (B3 (C10)) programs are related to checking validation accuracy [38]. The model splits the train data into training and validation data, and then provide the validation data by passing *validation_data=(x_val, y_val)* into the *fit()* method. The buggy model reported high accuracy for the validation dataset. There may exist an overlap between training data and validation data. But our approach would not indicate any symptom, as it does not support problems related to training and validation.

Table 8: The Symptoms Results from DeepDiagnosis

Dataset	DD - Symptoms									
	NS	UCS	SAS	DNS	ORS	LNDS	ANIS	VGS	IAS	ILS
***Stack Overflow* [7]**	15	1	5	1	2	0	1	0	1	0
GitHub [7]	2	1	2	1	1	0	0	1	1	0
Schoop *et al.* [8]	6	0	0	3	2	0	0	0	0	0
Blob [9]	5	7	2	0	0	0	10	10	0	0
Circle [9]	12	12	3	1	0	0	11	8	0	0
MNIST [9]	16	0	0	7	0	0	0	8	0	0
CIFAR-10 [9]	17	4	0	0	0	0	0	5	0	0

Both #43 (A2 (C10)), and #49 (A2 (C10)) programs are related to the dropout rate in the *Dropout* layer [38]. The idea of the dropout is to remove a certain percentage of neurons during iterations to prevent overfitting. The buggy model sets a high dropout rate = 0.8 which is more than the acceptable rate of 50%. Our approach is not able to provide a correct suggestion to fix the model. In our future work, we plan to investigate more hyperparameters such as the batch size, epoch, and dropout rate to handle the above models.

DD supports deep learning models of various structures, including convolutional neural networks (CNNs) and fully connected layers. But, Recurrent Neural Networks (RNNs) are not supported by our current reference implementation. Developers can extend our DD to support RNNs and other architectures.

UM only supports classification problems, in which the last layer is *softmax*. Otherwise, it reports false alarms. DL only supports numerical problems, and it does provide any suggestions on how to fix a detected problem. AT supports classification problems and does not support problems in the model architecture (i.e., loss function, activation function at last layer, and some APIs (e.g.,*fit_generator()*)). In terms of efficiency, AT takes longer to find a fix, as it tries all possible solutions until arriving at the correct one. In case it does not find an improvement, it marks the problem as unsolvable.

4.4 RQ4 (Ablation)

The "Ablation" column of Table 6 shows which procedure in Table 2 is used to report the symptom in each buggy model for SGS dataset. We found that *ExplodingTensor ()* detects 23 buggy models, *SaturatedActivation ()* detects 7, *DeadNode ()* reports 5, *OutofRange ()* detects 5, *UnchangeWeight ()* finds 2, *InvalidAccuracy ()* detects 2, *AccuracyNotIncreasing ()*, and *VanishingGradient ()* reports only one buggy model. Table 8 shows dataset names, and columns contain the number of symptoms, which were detected successfully by the corresponding procedure in Table 2. From Table 8, we found that *ExplodingTensor ()* detects 73 buggy models, *VanishingGradient ()* detects 32, *UnchangeWeight ()* finds 25, *AccuracyNotIncreasing () 22*, *DeadNode ()* reports 13, *SaturatedActivation ()* detects 12, *OutofRange ()* detects 5 , and *InvalidAccuracy ()* reports only two buggy models. Although the incorrect DNN models related to parameters and structures often manifest as numerical errors during training, DD provided further reasoning and categories of causes using these procedures, which can help quickly fix the bugs. Our study also found that data preparation is a frequently occurring issue and thus the *ImproperData()* procedure is frequently invoked. SGS benchmark does not have a very deep model that contains many layers. Thus we did not use *VanishingGradient ()* detector very frequently. On the other hand, *VanishingGradient ()* is invoked very frequently in AUTOTRAINER models, because this dataset has many layers using sigmoid and tanh as activation functions. However, when N layers use a Logistic activation function (like sigmoid or tanh), N small derivatives are multiplied together. Thus, the gradient decreases exponentially and propagates down to the input layer.

4.5 Results Discussions

We compared and contrasted three approaches [37, 44, 48] against our approach (DD). From Table 7, we found our approach detected more problems in the SGS dataset than AUTOTRAINER. Also, it detected fewer problems in AUTOTRAINER dataset than the AT approach. The reason is that our approach only reported the problem and solution if it detected one of 8 symptoms. On the other hand, AT inspects the model based on the training accuracy threshold [48].

For our evaluation, we used 188 normal models from [48]. From those, 78 are MNIST, 35 are CIFAR-10, 36 are Circle, and 39 are Blob. UM reported the message: "<Warning: Possible overfitting>" for 68 out 78 MNIST models. It reported the following message: "[<Error: Input data exceeds typical limits>]" for 35 out 35 CIFAR-10 models, because the training data is not in the range [-1, 1]. DL reported the message: "MDL: Model Does not Learn" for 4 out 34 Circle models and 16 out 39 Blob models. For all MNIST and CIFAR-10 models, DL reported different messages. AT checks if a model has training accuracy less than or equal to the threshold of 60%. To make a fair comparison between the approaches, we changed the training accuracy threshold to 100%. AT reported different symptoms for 10 out of 36 Circle, 5 out of 39 Blob models, and 2 models with problems out of the 78 MNIST models. Our approach reported one saturated symptom for 36 Circle, which is not supported in AT, reported 8 symptoms - 6 "saturated activations" and 2 the "accuracy is not increasing." For the MNIST model, our approach reported 37 symptoms - 35 "dead nodes" and one is a "numerical problem;" we investigated this model and found its accuracy is 20%. For CIFAR-10 models, DD reported 21 models with "dead node" out of 35 models. All detailed experiment results are publicly available [6].

4.6 Summary

DD significantly outperformed the baselines UM, DL, and AT in the SGS dataset (Tables 6 and 7). In particular, identified 46 out of 53 buggy models, correctly performed fault localization in 34 models, and reported symptoms for 37 of those. DD also provided a location to fix 28 out of 53 faulty models. Regarding total analysis time, DD outperformed AT because it does not require the training phase to finish to detect bugs. Also, DD uses a Decision Tree (Figure 2) approach to reduce the search space when mapping symptoms to their root causes.

Furthermore, DD is more comprehensive than prior work, as it supports several varieties and semantically related errors in classification and regression models. Also, DD supports 8 failure symptoms, while prior approaches support fewer (in Section 3).

Finally, DD does not support some APIs (e.g., *fit_generator()*) as we consider problems related to hyperparameters, for example, epoch, batch size, and dropout rate, as out of scope.

5 THREATS TO VALIDITY

External Threat: We have collected 53 real-world buggy DNN models from *Stack Overflow*, GitHub and 496 models from prior work [38, 44, 48]. These models cover a variety of failure symptoms and location to perform fixes; however, our dataset may not include all types of DNN APIs and their parameters. To mitigate the threat of behavior changes caused by the Extractor tool, we manually verified the accuracy of each model before and after their conversion. We used the Extractor to extract the source code from the 496 models from AUTOTRAINER [48]. In terms of execution time, different hardware configurations may offer varying response times. We mitigated this threat by executing our experiments several times and calculated their averages.

Internal Threat: When implementing Algorithm 1, Decision Tree (Figure 2), and Tables 2 and 3, we used the parameters defined by prior works [1, 10, 20, 37, 48]. These selected values may not work for some unseen examples. To mitigate this threat, we have validated these selected parameters against our benchmarks collected from a diverse set of sources [38, 44, 48]. For each of these benchmarks, our selected parameters work consistently well. Although we have carefully inspected our code, our implementation may still contain some errors. We manually constructed ground truths regarding fault location, failure symptoms, and location to fix for all the buggy models based on the data from the previous research [38, 44, 48]. This process may have introduced errors.

6 RELATED WORK

Fault localization in Deep Neural Networks: The recent increase in the popularity of deep learning apps has motivated researchers to adapt fault localization techniques to this context. With the intent of validating different parts of DL-based systems and discovering faulty behaviors. The goal of fault localization is to identify suspicious methods and statements to isolate the root causes of program failures and reduce the effort of fixing the fault [36]. Wardat *et al.* [44] presented an automatic approach for fault localization called DeepLocalize. It performs dynamic analysis during training to determines if a target model contains any bugs. It identifies the root causes by catching numerical errors during DNN training. While DeepLocalize focuses on identifying bugs and faults based on numerical errors, DeepDiagnosis aims to perform fault localization beyond that scope. Furthermore, our approach can report symptoms and provide actionable fixes to a problem.

DEBAR [49] is a static analysis approach that detects numerical bugs in DNNs. DEBAR uses two abstraction techniques to improve its precision and scalability. DeepDiagnosis uses dynamic analysis to localize faults and report symptoms of a model during training. In contrast, DEBAR uses a static analysis approach to detect numerical bugs with two abstraction techniques.

Schoop *et al.* [38] proposed UMLUAT, a user interface tool to find, understand and fix deep learning bugs using heuristics. It enables users to check the structure of DNN programs and model behavior during training. Then, it provides readable error messages to assist users in understanding and fixing bugs. Section §4 shows the comparison between UMLUAT [38] and DeepDiagnosis. DeepDiagnosis is more comprehensive, efficient, and effective than UMLAUT, which only supports classification models.

DeepFault [15] is an approach that identifies suspicious neurons of a DNN and then fixes these errors by generating samples for retraining the model. DeepFault is inspired by spectrum-based fault localization. It counts the number of times a neuron was active/inactive when the network made a successful or failed decision. It then calculates a suspiciousness score such as the spectrum-based fault localization tool Tarantula. In contrast, DeepDiagnosis focuses on identifying faults and reporting different types of symptoms for structure bugs.

Bug Repair in Deep Neural Networks: Zhang *et al.* [46] proposed Apricot, an approach for automatically repairing deep learning models. Apricot aims to fix ill-trained weights without requiring additional training data or any artificial parameters in the DNN. MODE [30] is a white-box approach that focuses on improving the model performance. It is an automated debugging technique inspired by state differential analysis. MODE can determine whether a model has overfitting or under-fitting problems. Compared with MODE and Apricot, which focus on training bugs (e.g., insufficient training data), DeepDiagnosis focuses on structure bugs (e.g., activation function misused).

Zhang *et al.* [48] introduced AUTOTRAINER, an approach for fixing classification problems. Zhang *et al.* define five symptoms, and provide a set of possible solutions to fix each one. Once AUTOTRAINER detects a problem, it tries the candidate solutions, one by one, until it addresses the problem. If none of the solutions fix the problem, it reports a failure message. The evaluation used six popular datasets and showed that AUTOTRAINER detects and repairs the models based on a specific threshold. AUTOTRAINER was able to improve the accuracy for all repairing models on average 47.08%. DeepDiagnosis analyzes the model's source code during the training phase to localize the bug. DeepDiagnosis supports eight symptoms, while AUTOTRAINER supports five. DeepDiagnosis does not perform automated fixes, but it provides actionable recommendations that developers can follow. AUTOTRAINER tries all possible strategies in its search space to fix a problem and outputs whether or not the fix was successful. In contrast, DeepDiagnosis uses a decision tree to reduce the solution search space, thus saving time and computational resources. In summary, the goals of DeepDiagnosis and AUTOTRAINER are different; DeepDiagnosis focuses on fault localization while AUTOTRAINER on automatically repairing a model.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces a dynamic analysis approach called DeepDiagnosis that a non-expert can use to detect errors and receive useful messages for diagnosing and fixing the DNN models. DeepDiagnosis provides a list of verification procedures to automatically detect 8 types of common symptoms. Our results show that DeepDiagnosis can successfully detect different types of symptoms and report actionable changes. It outperforms the state of the art tool such as UMLUAT and DeepLocalize, and it is faster than AUTOTRAINER for fault localization and provide suggestions to fix the issue.

We have identified several future work directions. First, we would like to extend our approach to support additional model types, failure symptoms, and automatic repair. Second, we would like to conduct studies on how to improve DNN bug repair on non-functional bugs such as fairness bugs [11, 12]. Third, we would like to extend our approach to support additional types of bugs in different stages of the ML pipeline [13]. Lastly, we would like to explore how to leverage our findings to improve the performance of AutoML models [35].

8 ACKNOWLEDGMENT

This work was supported in part by the US National Science Foundation (NSF) through grants CNS-21-20448, and CCF-19-34884. All opinions are of the authors and do not reflect the view of sponsors.

REFERENCES

[1] 2015. https://cs231n.github.io/neural-networks-3/. [Online; accessed 20-Aug-2020].

[2] 2016. How to prepare a dataset for Keras? https://stackoverflow.com/questions/31880720/. [Online; accessed 19-Aug-2020].

[3] . 2020. Manifold. https://github.com/uber/manifold.

[4] . 2020. Tensorwatch. https://github.com/microsoft/tensorwatch.

[5] . 2020. Visdom. https://github.com/fossasia/visdom.

[6] 2021. https://github.com/DeepDiagnosis/ICSE2022. [Online; accessed 12-August-2021].

[7] 2021. https://github.com/Wardat-ISU/DeepLocalize. [Online; accessed 12-Aug-2021].

[8] 2021. https://github.com/BerkeleyHCI/umlaut. [Online; accessed 12-Aug-2021].

[9] 2021. https://github.com/shiningrain/AUTOTRAINER. [Online; accessed 12-August-2021].

[10] Amazon. 2017. Amazon SageMaker. https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html.

[11] Sumon Biswas and Hridesh Rajan. 2020. Do the Machine Learning Models on a Crowd Sourced Platform Exhibit Bias? An Empirical Study on Model Fairness. In *ESEC/FSE'2020: The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States).

[12] Sumon Biswas and Hridesh Rajan. 2021. Fair Preprocessing: Towards Understanding Compositional Fairness of Data Transformers in Machine Learning Pipeline. In *ESEC/FSE'2021: The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece).

[13] 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).

[14] François Chollet. 2019. Writing your own callbacks. https://keras.io/guides/writing_your_own_callbacks/. [Online; accessed 20-Apil-2021].

[15] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: fault localization for deep neural networks. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 171–191.

[16] Utku Evci. 2018. Detecting dead weights and units in neural networks. *arXiv preprint arXiv:1806.06068* (2018).

[17] Francois Chollet. 2015. Keras documentation. https://keras.io/.

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

[19] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.

[20] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[21] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. 2009. A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the sigchi conference on human factors in computing systems*. 649–658.

[22] Guido van Rossum. 2019. Module difflib. https://github.com/python/cpython/blob/3.9/Lib/difflib.py.

[23] Caglar Gulcehre, Marcin Moczulski, Misha Denil, and Yoshua Bengio. 2016. Noisy activation functions. In *International conference on machine learning*. PMLR, 3059–3068.

[24] Arnekvist Isac, Carvalho J Frederico, Danica Kragic, and Johannes Andreas Stork. 2020. The effect of Target Normalization and Momentum on Dying ReLU. In *The 32nd annual workshop of the Swedish Artificial Intelligence Society (SAIS)*.

[25] 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)*.

[26] 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).

[27] Andrew Janowczyk and Anant Madabhushi. 2016. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. *Journal of pathology informatics* 7 (2016).

[28] Steven W Knox. 2018. *Machine learning: a concise introduction*. Vol. 285. John Wiley & Sons.

[29] Shumin Kong and Masahiro Takatsuka. 2017. Hexpo: A vanishing-proof activation function. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2562–2567.

[30] 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.

[31] D Mané et al. 2015. TensorBoard: TensorFlow's visualization toolkit.

[32] Martin Abadi et al. 2015. TensorFlow: large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/.

[33] John Miller and Moritz Hardt. 2018. Stable recurrent models. *arXiv preprint arXiv:1805.10369* (2018).

[34] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. 2018. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics* 19, 6 (2018), 1236–1246.

[35] Giang Nguyen, Johir Islam, Rangeet Pan, and Hridesh Rajan. 2022. Manas: Mining Software Repositories to Assist AutoML. In *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.

[37] Eldon Schoop, Forrest Huang, and Björn Hartmann. 2020. SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–10.

[38] 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*.

[39] Shanqing Cai. 2017. Debug TensorFlow Models with tfdbg. https://developers.googleblog.com/2017/02/debug-tensorflow-models-with-tfdbg.html.

[40] David Sussillo and LF Abbott. 2014. Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558* (2014).

[41] Hong Hui Tan and King Hann Lim. 2019. Vanishing gradient mitigation with deep learning neural network optimization. In *2019 7th International Conference on Smart Computing & Communications (ICSCC)*. IEEE, 1–4.

[42] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[43] Matthew Veres and Medhat Moussa. 2019. Deep learning for intelligent transportation systems: A survey of emerging trends. *IEEE Transactions on Intelligent transportation systems* 21, 8 (2019), 3152–3168.

[44] 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*.

[45] Bing Xu, Ruitong Huang, and Mu Li. 2016. Revise saturated activation functions. *arXiv preprint arXiv:1602.05980* (2016).

[46] Hao Zhang and WK Chan. 2019. Apricot: a weight-adaptation approach to fixing deep learning models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 376–387.

[47] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*. 272–281.

[48] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *ICSE'21: The 43nd International Conference on Software Engineering*.

[49] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 826–837.