DeepLocalize: Fault Localization for Deep Neural Networks

Mohammad Wardat
Department of Computer Science
Iowa State University
2434 Osborn Dr
Ames, IA, 50011, USA
wardat@iastate.edu

Wei Le
Department of Computer Science
Iowa State University
2434 Osborn Dr
Ames, IA, 50011, USA
weile@iastate.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
2434 Osborn Dr
Ames, IA, 50011, USA
hridesh@iastate.edu

Abstract— **Deep neural networks (DNNs) are becoming an integral part of most software systems. Previous work has shown that DNNs have bugs. Unfortunately, existing debugging techniques don't support localizing DNN bugs because of the lack of understanding of model behaviors. The entire DNN model appears as a black box. To address these problems, we propose an approach and a tool that automatically determines whether the model is buggy or not, and identifies the root causes for DNN errors. Our key insight is that historic trends in values propagated between layers can be analyzed to identify faults, and also localize faults. To that end, we first enable dynamic analysis of deep learning applications: by converting it into an imperative representation and alternatively using a callback mechanism. Both mechanisms allows us to insert probes that enable dynamic analysis over the traces produced by the DNN while it is being trained on the training data. We then conduct dynamic analysis over the traces to identify the faulty layer or hyperparameter that causes the error. We propose an algorithm for identifying root causes by capturing any numerical error and monitoring the model during training and finding the relevance of every layer/parameter on the DNN outcome. We have collected a benchmark containing 40 buggy models and patches that contain real errors in deep learning applications from *Stack Overflow* and *GitHub*. Our benchmark can be used to evaluate automated debugging tools and repair techniques. We have evaluated our approach using this DNN bug-and-patch benchmark, and the results showed that our approach is much more effective than the existing debugging approach used in the state-of-the-practice *Keras* library. For 34/40 cases, our approach was able to detect faults whereas the best debugging approach provided by *Keras* detected 32/40 faults. Our approach was able to localize 21/40 bugs whereas *Keras* did not localize any faults.**

Index Terms—**Deep Neural Networks, Fault Location, Debugging, Program Analysis, Deep learning bugs**

I. Introduction

Deep neural networks are a class of machine learning algorithms that have gained significant popularity in recent years due to their remarkable success in tasks that defy traditional algorithm techniques. A deep neural network can be thought of as a graph where nodes, called *neurons*, are functions with adjustable weights. The neurons of a DNN are organized in layers and edges feed output from a neuron to neurons in the next layer, and eventually to the last layer called the output layer. During the training step, each training input is passed through the network to produce output. This output is compared to the expected output. The difference between the actual output and the expected output, measured using a function called the *loss function*, is then used to adjust the weights of the neurons in the layers using a process called *back propagation*. DNNs are utilized in various software systems to make decisions. Thus, software engineering for DNNs has become essential.

To aid integration of the DNN in software systems, a number of developers have produced industrial-strength libraries and frameworks such as *TensorFlow* [1], *Cafe* [2], *MXNet* [3], *PyTorch* [4], *Theano* [5] and *Keras* [6] to assist the programmers in designing reliable deep learning applications. Recent work has shown that applications that use DNN have bugs [7]–[9]. Same group of researchers have also studied repair strategies for DNN [10]. Zhang *et al.* [8] describe the challenges and limitations in detecting and localizing the bugs in the DNN model, and indicate that current approaches are not effective to examine the state of the model at some point, like the regular programs. Islam *et al.* observe that DNN bug fix patterns are distinctive compared to traditional bug fix patterns, and that DNN bug localization is among the major challenges faced by developers when fixing bugs [10].

Despite the growing number of software debugging techniques such as automated bug repair [11], [12], fault localization [13], [14], delta debugging [15], and slicing [16], these techniques are still not applicable to identify the bugs in the DNN models and identify the faulty statements that cause the problem at a particular layer in the model. Regular software programs and the DNN models are fundamentally different with respect to fault and fault detection. For example, regular software programs are tested by comparing the actual output and the expected output. If actual output doesn't match the expected output, then we consider the program has a bug. On the other hand, the DNN-based software has a complex structure, and it is learning from a training dataset. If the DNN produces incorrect classification during training, we call it failure case, it is not necessarily that DNN contains a bug, because a DNN model cannot guarantee 100% correct classifications. Furthermore, the logic of a regular program is represented in terms of control flow, while DNN programs use weights between neurons and different kinds of activation functions

for similar purposes. These differences make debugging and testing of software that deploys DNNs challenging.

Traditional practices for debugging uses aides such as print statements, breakpoints, and tracing the failing test case. These manual debugging processes take a long time and effort from developers [17]. Researchers have proposed several automated fault localization techniques [13], [18], [19]. These techniques are used to locate the root causes and understand the faulty states. Unfortunately, current automated fault localization techniques cannot be applied directly to DNN since existing techniques are not able to identify plausible and distinct root causes for unexpected behavior (known as failure) in DNNs.

To overcome these challenges, this work introduces a white box based fault localization technique for DNNs. Our approach requires the source code of the DNN model and the training data. Given the source code, our approach enables dynamic trace collection for DNN. We propose two techniques. The first technique, inspired by [20], translates the code into an intermediate form which we call *imperative representation of the DNN*. The purpose of the imperative representation is to make certain (ensure) that internal states of the DNN are observable, thus our method uses a *white box approach*. This conversion to an imperative representation allows us to insert probes that enable dynamic analysis over the traces produced by the DNN while it is being trained on the training data. The second technique uses a novel callback mechanism to insert probes that also achieve the same purpose. We then conduct dynamic analysis over the traces to identify the faulty layer or hyperparameter that causes the error. We also propose an algorithm to identify root causes by capturing any numerical error and monitoring the model during training and finding the relevance of every layer/parameter on the DNN outcome.

We have implemented our approach as an extension of the widely used *Keras* library for deep learning. To evaluate, we have collected a benchmark containing 40 buggy models and patches that contain real errors in deep learning application from *Stack Overflow*, and *GitHub*. Our benchmark can be used to evaluate automated debugging tools and repair techniques. We compare our approach with three built-in mechanisms for debugging in the *Keras* library, the state-of-the-art in DNN libraries. These mechanisms were *TerminateOnNaN()*, *EarlyStopping ('loss')*, and *EarlyStopping ('accuracy')*. We have evaluated our approach using this DNN bug-and-patch benchmark, and the result shows that our approach is much more effective than the existing debugging approach used in the state-of-the-practice *Keras* library. For 34/40 cases, our approach was able to detect faults whereas the best debugging approach provided by *Keras* detected 32/40 faults. Our approach was able to localize 21/40 bugs whereas *Keras* did not localize any faults. In summary, this paper makes the following contributions:

- We propose the *first fault localization approach for DNNs* including *callback and translation mechanisms for collecting dynamic traces* and a *localization algorithm*.
- We have built a *DNN bug-and-patch benchmark* with 40 different types of buggy models from *Stack Overflow* and

GitHub. This benchmark serves as the ground truth to evaluate our approach. We also hope it can serve other researchers to validate their debugging and repair tools. This benchmark is available from *GitHub* [21].
- Our results show that our approach can effectively and efficiently identify 34 out of 40 buggy model and determine the root causes for 21 out of 40 buggy model.

Outline: §II motivates our approach. §III presents our dynamic trace collection, faulty detection and localization algorithms. §IV presents evaluation. §V discusses the threats to validity, §VI discusses related works and §VII concludes.

II. A MOTIVATING EXAMPLE

Listing 1 shows a simple example from *Stack Overflow* [22] to motivate our work. In this example, the developer is constructing a sequential model at line 1, adding a dense input layer at lines 2-3, adding a dense hidden layer at lines 4-5, compiling the model to convert it to a graphical form on line 6, and training this model on line 7. A dense layer is a layer in a DNN where each neuron is connected to neurons in the next layer. This DNN did not learn during training and in the post, the developer asked why the model achieved low accuracy. This DNN has a two problems. First, it handles a classification problem, and thus `categorical_crossentropy` should be used as a loss function at line 6 instead of `mean_squared_error`. Second, the user has not added activation functions for the first two layers at lines 2 and 4.

Listing 1: DNN is not learning [22]

```
1  model = Sequential()
2  layer1 = Dense(30, input_shape=input_shape,
       kernel_initializer=RandomNormal(stddev=1)
       , bias_initializer=RandomNormal(stddev=1)
       )
3  model.add(layer1)
4  layer2 = Dense(10, kernel_initializer=
       RandomNormal(stddev=1), bias_initializer=
       RandomNormal(stddev=1))
5  model.add(layer2)
6  model.compile(optimizer=SGD(lr=3.0), loss='
       mean_squared_error', metrics=['accuracy'
       ])
7  model.fit(x_train, y_train, batch_size=10,
       epochs=30, verbose=2)
```

Keras provides a set of callback methods [23] to give the developers information about internal status of the training process. Specifically, we can use `TerminateOnNaN()` to monitor the loss and terminate the training when the loss becomes `NaN`. We can use `EarlyStopping()` to monitor the loss or accuracy and stop if there is no improvement. We can pass a callback method as a parameter to the `fit()` method. For Listing 1, when using `TerminateOnNaN()`, `EarlyStopping('loss')`, `EarlyStopping('accuracy')`, and the union of the three *Keras* methods, the training was terminated after 1.20, 12.21, 34.90 and 1.16 seconds respectively. Once the training is stopped, *Keras* prints the epoch and the iteration number. Unfortunately, such information cannot answer the developer's question and
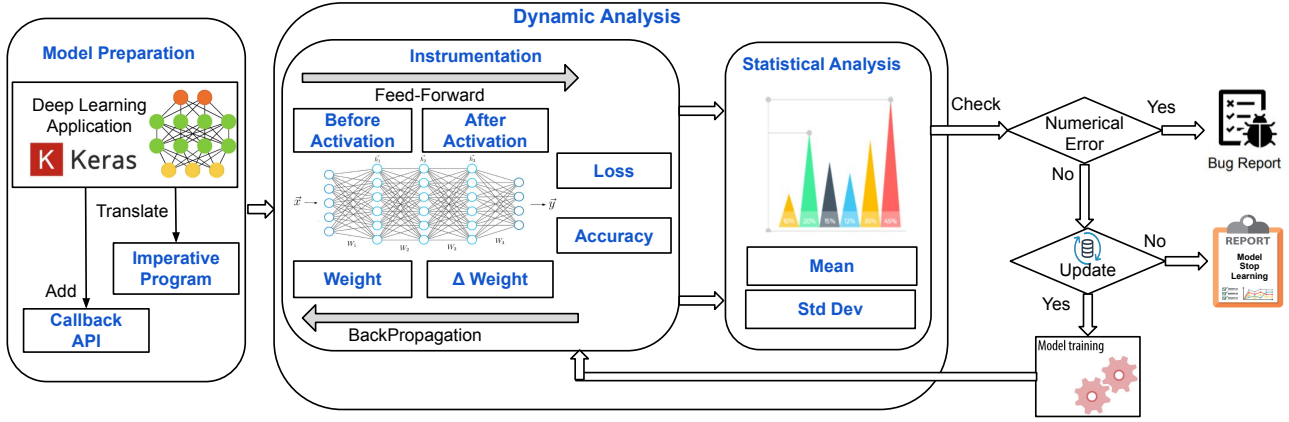
Fig. 1: An Overview of DeepLocalize. Left component shows two alternatives (callback and translation) for preparing models to collect dynamic traces. Middle component collects and analyzes these traces. Right component detects/localizes bugs.

indicate which layer or hyperparameter prevented the model from learning.

Our approach reports that the program has a bug after 0.14 seconds using our tool and 2.14 seconds using our callback function. In addition, we report that the bug is located in the back propagation stage of layer 2 at line 4. Therefore, the message gives the developer hint that the issue in the parameter of layer 2, since the description of the message indicates the stage of the problem, the developer can quickly determine previous calculation that causes the problem, which is the loss function in our example. Compared to the existing methods in *Keras*, our fault localization uses less time and provides a report stating the layer that the bug is located at.

III. APPROACH

A. An Overview

Figure 1 provides an overview of our work. In the first step, we prepare the DNN model to collect dynamic traces. We propose two mechanism for this. Our first approach translates the deep learning program into an imperative program [20], [24]. Probes are inserted in this imperative program to capture and save model variables such as weights, learning rate gradients during training. Our second approach uses a callback mechanism, and passes a specialized callback method as a parameter during model training (to the `fit()` method). This custom callback function allows the developers to capture and save model variables. We record the key values during both the feed-forward and backward propagation phases. During the training, an online statistical analysis is performed to compare the status of the program with the error conditions we defined. Finally, we report if the program contains a bug, and in which layer and phases the bug exists that prevented learning.

B. Model Preparation

Directly analyzing a deep learning program, e.g. one shown in Listing 1 is hard, as the DNN libraries provide blackbox APIs and it is hard to trace important values during training.

To use our approach, the developer either write extra code to instrument DNN training inside `fit()` function or rewrite their code into an imperative form. For our second approach, we identified a list of the *Keras* library APIs that are important for training and implemented models/simplified versions of these API calls, following the machine learning literature [25]–[27] and the *Keras* documentation [28]. We inserted the probes to these library models so that the analysis can observe the internal behaviors of DNNs during training.

The callback-based approach of dynamic trace collection is implemented by overriding the `keras.callbacks.Callback` class. Since our work is focusing on monitoring during the training phase, we override the method called (`on_train_batch_end(self, batch, logs=None)`). This overridden method invokes Algorithm 1 after each batch of training. To use this method, the developer needs to pass this custom callback function as a parameter to the `fit()` function.

Second imperative approach, as shown in Table I on the left, to build a training model, a DNN program typically starts with creating a sequential model (line 4), then add all layers (lines 5–11) and optimizations (line 13), and finally call `compile` and `fit` at lines 14–15. On the right, we show the imperative programs using our library models. First, lines 1, 2, 7 and 10 are not changed. Second, lines 5, 8, and 11 show the conversions for the Dense layer. In our code, we added a name for the layers, e.g., see name = "FC1" so that we can report in which layer the fault is located. We inserted instrumentation in the fit function (line 15) to observe the model variables.

Currently, our translation tool supports the `Dense`, `Dropout`, `Maxpooling`, `Convolution`, `BatchNormalization`, and `Padding` layers. Also, it supports popular optimization methods, losses, and activation functions. The translation from a DNN program to the program that uses our library models is currently done manually. The *Keras* library is being rapidly evolved resulting in a large number of releases [10]. Due to the library versioning and the frequently changing API signatures, it is hard for our tool to remain compatible

TABLE I: Translation from *Keras* Code to an Imperative program

No	*Keras* Code	Imperative Program
1	batch_size = 1	batch_size = 1
2	nb_epoch = 3	nb_epoch = 3
3		+ lr = 0.001
4	model = Sequential()	myModel = mySequential()
5	- model.add(Dense(units=50, activation='relu', input_dim=128))	+ myModel.insert(myDense(num_inputs=128, num_outputs=50, lr_rate=lr, name='FC1'))
6		+ myModel.insert(ReLu())
7	model.add(Dropout(0.2))	myModel.insert(myDropout(0.2))
8	- model.add(Dense(units=50, activation='relu'))	+ myModel.insert(myDense(num_inputs=50, num_outputs=50, lr_rate=lr, name='FC2'))
9		+ myModel.insert(ReLu())
10	model.add(Dropout(0.2))	myModel.insert(myDropout(0.2))
11	- model.add(Dense(1,activation ='softmax'))	+ myModel.insert(myDense(num_inputs =50, num_outputs =1, lr_rate =lr, name='FC3'))
12		+ myModel.insert(Softmax())
13	- Adam = optimizers.Adam()	
14	- model.compile(loss ='binary_crossentropy', optimizer =Adam, metrics=['acc'])	+ myModel.myCompile(loss='binary_crossentropy', optimizer='Adam', metrics=['acc'])
15	- model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=3, verbose=1, validation_data=(X_test, Y_test))	+ myModel.fit_instrument(X_train, Y_train, batch_size=batch_size, epoch=3)
16	- model.evaluate(X_test, Y_test, verbose =1)	+ myModel.myEvaluate(X_test, Y_test, 200)

This table is showing all of the changes to translate from *Keras* code to Our tool code. The color in each row indicates the change type: Green + added a new line, Red - removed an existing line.

with the *Keras* library.

C. Instrumentation

The training of DNN has two phases: feedforward and backpropagation. In the feedforward stage, we observe and monitor: (1) the training data including both input and label; (2) the results after applying forward function; (3) the results after applying the activation function; (4) the loss value and (5) the accuracy. All values are collected in each iteration during training.

The second stage is the backpropagation, and it is used to adjust the weight based on the errors obtained from the feedforward stage. Backpropagation uses the gradient descent method to update the weights and minimize the errors. It is started from the output layer, and the result of the output layer is reused to compute the gradient for the previous layer until it reaches the input layer. Different optimizations can be applied during backpropagation. In the backpropagation stage, we can observe and monitor: (1) the update of weights, (2) the update of bias values, and Δ weights from applying the gradient descent for each layer.

The instrumentation is inserted in the fit() function we implemented for modeling the *Keras* fit() function. It is executed automatically when the DNN program runs during training.

D. Statistical Analysis to Detect Suspect Behavior

Next, we discuss our approach for statistical analysis to detect suspect behavior of the DNN during training. We analyze three variables: the learning rate, the input data, and activation/loss functions.

1) Incorrect learning rate: Backpropagation is important to fine-tune the weights based on the loss value obtained from the loss function. The learning rate has an effect on the weight updates during the backpropagation process. During the backpropagation process, the learning library computes gradient descent iteratively. Our key insight is that the mean and standard deviation of the weights in the correct model are continuously changing during the training process. In contrast,

the mean and standard deviation of the weights in the buggy model are constant. If there is a problem in the learning rate, we can detect it from the output of the gradient for the output layer. Figure 2 and Figure 3 show an example of this behavior. In Figure 2 the weight varies as it should in a correct model, whereas in Figure 3 the weight is constant indicating a potential bug. To utilize this insight, we compute the mean and the standard deviation for the output of the gradient as well as the weight parameter at each layer.

2) Incorrect input data: In some cases, the training data are not properly normalized. For example, in the MNIST model, the pixel should be in the range [-1, 1] and not [0, 255]. Also, training data may have NaN value, and developers forget to invoke the assert function to check if there is an NaN or not. In the forward stage, we retrieve the output before/after applying the activation function for each year. We then compute the mean and standard deviation for the output at each layer. We will check if the output of the first layer after/before applying activation has numerical error such as NaN or Inf. Our second approach for detecting this kind of error is to calculate how frequent the mean of the output for the first layer is equal to zero. Once we observe abnormal values, we will report in which layer and whether it is before or after activation function that the error occurs.

3) Incorrect activation/loss functions: After finishing the forward stage, we compute loss and accuracy. There are two indicators that the model has a problem. First, if one of the two metrics has a numerical error like inf or NaN. Second, the loss starts increasing instead, or the accuracy starts decreasing after certain iterations.

E. DNN Fault Localization Algorithm

Algorithm 1 presents our DNN fault localization algorithm. At its core, it augments the DNN learning algorithm with analysis and error checking inserted during learning. It serves three purposes: (1) determining whether the deep learning program contains a bug; (2) reporting the fault location, in which layer and which phases (feed forward and backward propagation), the deep learning program has a bug; (3) reporting failure
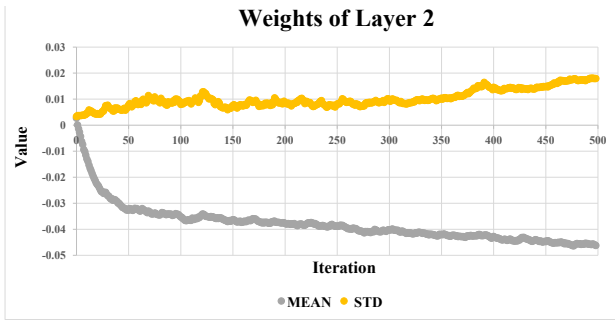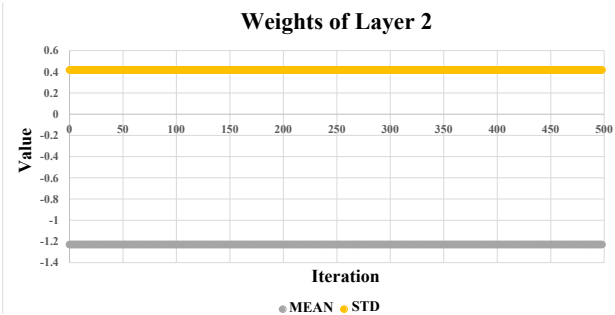
Fig. 2: Weight of correct model

Fig. 3: Weight of buggy model

No	Statement	Abbreviation
1	Error Before Activation	EBA
2	Error After Activation	EAA
3	Error in Loss Function,	ELF
4	Error in Accuracy Function	EAF
5	Error Backward in Weight	EBW
6	Error Backward in Δ Weight	EBDW
7	Model Does not Learn	MDL
8	Correct Model	CM

determine if there is any numerical error at line 13. As shown int Table II, the ELF indicates Error in Loss Function. If the error is not detected here, we save the loss value for each iteration at line 14. At line 15, we compute the accuracy and check if there is a numerical error from accuracy at Lines 16-18. If the error is not detected here, we save the accuracy value during training at Line 19. At lines 20-22, the algorithm checks if loss is not decreasing and accuracy is not increasing value for a long time. In both cases, The algorithm computes the slope to compare the loss/accuracy for current step with the loss/accuracy at a step which is at least num_steps behind the current step. In this case, the algorithm reports a message MDL to indicate that the model does not learn. Otherwise, the training continues.

2) Backpropagation phase: During this stage, the algorithm collects the weight and Δ weight for each layer in each iteration. At line 25, the Weight and Δ Weight are the output of a back-propagation. At line 26, the algorithm invokes the ANA() procedure and pass Δ weight to check if there is any numerical error. The algorithm will print error message if there is any error in the Δ weight and determines which layer causes this issue (Line 26). In the same way, at line 27, the algorithm can determine if there is an issue in the weight in each layer by invoke the ANA() procedure. If the procedure decide that there is an issue in the weight, then the algorithm will return message to indicate there is bug.

Finally, if there are no issues in the model, the algorithm will terminate after finishing training at line 32 and print this message Correct Model (CM).

ANA() is invoked at lines 7, 9, 26 and 27 to determine if the error occurs based on the current values obtained from the instrumentation. When the DNN does not learn, there can be the following symptoms: (1) the update for weight, Δ weight are incorrect, (2) the loss or accuracy is not measured on the correct scale, and (3) the loss does not decease, and the accuracy does not increase after the number of iterations. The check is conducted in ANA().

This procedure takes three parameters: input value, layer number and location. Since ANA() is called at different locations in the code, the location tracks whether the value comes from feed-forward (FW), backward (BW) propagation, or weight (WT). Line 2 defines a set of lists to store the mean value from each location for each layer. Line 3 computes the mean value for the input. At line 4, the procedure will check if the mean reports NaN or inf. Also, the procedure will check if the mean equal to zero at line 5; if yes, then we compute

information, in which iteration, the learning is stopped. It takes as input the training data set (including input and labeling), the translated imperative program, as well as the DNN parameters (the batch size and the epoch). If the bug is found, the output is a message including the fault location and failure information for the bug.

At line 1, we define two lists to store the values of loss and accuracy in each iteration. Line 2 represents how many iterations we make through the whole training dataset. During the training process, the training dataset is divided into several smaller batches. For example, if the model has 2000 training examples divided into 500 batches, then the model needs 4 iterations to complete one epoch. Line 3 shows the division of the batch size. Lines 4-29 run for each batch in the training dataset. At lines 5–11, the algorithm performs dynamic analysis on the forward stage, and at lines 24–29, it performs analysis on the backward stage. In our callback function, the override method `on_train_batch_end(self, batch, logs=None)` will execute at the end of each batch, and performs dynamic analysis on the forward/backward stage, after retrieving the value of each layer before/after activation function, loss, accuracy, updating weight and gradients.

1) Feedforward phase: At line 6, the algorithm computes the output of a feed-forward layer before applying the activation function. At line 8, we compute the output after applying the activation function. Then, we invoke ANA() procedure to determine if the output contains a numerical error. As shown in Table II, the message EBA indicates Error Before Activation, EAA indicates Error After Activation, and L represents the faulty layer number. At line 12, we compute the loss, and

how frequent zero occurs for all values for each layer. If the number of zeroes is greater than a threshold (line 7), the error is detected and the procedure will return true.

At line 11, we store the mean value in the list. Finally, the procedure will return the last N element from the list as slice to check if the mean value for last N iterations is changed or not. From Figure 2 and Figure 3, we observe the model continues to learn if the mean value is changing in each iteration. This procedure returns true if there is a numerical error; otherwise, it return false.

Algorithm 1: DNN Fault Localization

input : Training data (input, label), batchsize, epochs, imperative program
output: A message regarding fault location and failure information

1   $LossList \leftarrow []$; $AccuList \leftarrow []$
2   **for** $e \leftarrow 0$ **to** $epoches$ **do**
3   **for** $i \leftarrow 0$ **to** $Length(input)$ **Step** $batchsize$ **do**
4       $X \leftarrow input[i]$; $Y \leftarrow label[i]$
5   **for** $L \leftarrow 0$ **to** $Length(Layers)$ **do**
6         $V_1 \leftarrow Layer[L].Forward(X)$
7   **if** $ANA(V_1, L, FW)$ **then** **return** EBA, L
8         $V_2 = Layer[L].Activation(V_1)$
9   **if** $ANA(V_2, L, AF)$ **then** **return** EAA, L
10         $X \leftarrow V_2$
11   **end**
12       $Loss \leftarrow ComputeLoss(V_2, Y)$
13   **if** $Loss$ is equal to NaN OR inf **then** **return** ELF
14       $LossList.append(Loss)$
15       $Accuracy \leftarrow ComputeAccuracy(V_2, Y)$
16   **if** $Accuracy$ is equal to NaN OR inf OR 0 **then**
17   **return** EAF
18   **end**
19       $AccuList.append(Accuracy)$
20   **if** $NOTDecreasing(AccuList)$ && $NOTIncreasing(LossList)$ **then**
21   **return** MDL
22   **end**
23       $dy \leftarrow Y$
24   **for** $L \leftarrow Length(Layers)$ **to** 0 **do**
25         $V_3, W[L] \leftarrow Layer[L].Backward(dy)$
26   **if** $ANA(V_3, L, BW)$ **then** **return** EBDW, L
27   **if** $ANA(W[L], L, WT)$ **then** **return** EBW, L
28         $dy \leftarrow V_3$
29   **end**
30   **end**
31   **end**
32   **return** CM

---

1 **Function** ANA(*input, LayerNo, Location*)**:**
2    $meanList[LayerNo, Location] \leftarrow []$
3    $meanValue \leftarrow math.mean(input)$
4   **if** $meanValue$ is equal NaN OR inf **then** **return** True
5   **if** $meanValue == 0$ **then**
6      $FreqZero[LayerNo, Location] += 1$
7   **if** $FreqZero[LayerNo, Location] \geq threshold$ **then**
8   **return** True
9   **end**
10   **end**
11    $meanList[LayerNo, Location].append(meanValue)$

12    $slicing \leftarrow meanList[LayerNo, Location][-N:]$
13   **if** $All(elem == slicing[0] \ for \ elem \ in \ slicing)$ **then**
14   **return** True
15   **end**
16   **return** False

TABLE III: *Keras* result VS Imperative result

			Keras			Our Tool		
			Runtime	Result		Runtime	Result	
Post #	Epoch	Iteration	[sec]	Loss	Accuracy	[sec]	Loss	Accuracy
48385830	30	60000	489.67	NaN	0.10	1429.60	NaN	0.10
31556268	1000	1	5.58	0.50	0.50	20.99	0.50	0.50
50306988	5	200	1.41	0.71	0.50	0.17	0.69	0.50
48251943	17	500	5.40	0.37	–	2.10	0.30	–
38648195	20	48000	123.21	0.23	0.75	385.40	NaN	0.65
33969059	20	10000	129.85	349713063.30	–	1272.30	260179130.95	–
55328966	10	49999	104.95	2.30	0.10	8025.54	2.29	0.11
34311586	20	6	1.23	0.67	–	0.11	0.50	–
31880720	3	20000	181.30	7.67	0.50	529.29	16.12	0.50
39525358	150	13	3.15	0.67	0.62	5.16	0.67	0.61
48934338	1000	50	10.17	1358.22	0.00	28.58	787.75	0.00
47724077	50	32561	39.26	49.70	0.68	93.46	3.38	0.70
59278771	200	135	18.91	1.10	0.35	14.86	1.06	0.34
41372874	20	239	3.19	0.15	1.00	6.12	0.00	1.00
44066044	100	21	3.86	9.55	0.38	0.89	9.55	0.00
51930566	50	75	4.04	0.51	0.67	1.87	0.48	0.64
45442843	100	200	2.74	0.50	0.50	23.34	0.51	0.50
31627380	10	712	5.75	1.29	0.59	8.31	1.46	0.63
58609115	10	442	2.63	26169.69	0.09	13.10	4.05	0.02
50481178	50	200	11.82	0.02	0.99	6.13	0.01	0.98

- RQ3 (Limitation): In which cases does our technique fail to report the bug and localize the faults?

A. Experimental setup

1) Implementation: To perform our experiments and evaluation, we implemented our techniques using Python and *Keras*. Our translation-based tool supports the `Dense`, `Dropout`, `Maxpooling`, `Convolution`, `BatchNormalization`, and `Padding` layers. Also, it supports popular optimization methods, losses, and activation functions. We followed the machine learning references [25]–[27] and used the *Keras* documentation to implement simplified and instrumented versions of `compile()`, `sequential()` and `fit()` function. Our callback-based tool supports all of the layers and optimization methods supported by *Keras*.

We set $threshold = 1/4 * No.$ Iteration and N= 50 at line 6 and 12 in ANA() function respectively for both our tool and callback function. Based on our empirical experience, these settings helped best in detecting bugs during training.

IV. Evaluation

In this section, we aim to answer the following research questions:

- RQ1 (Validation): Can our technique find bugs in deep learning programs effectively?
- RQ2 (Comparison): How effectively and how fast can our technique localize the faults compared to existing methodology in the *Keras* library?

2) Benchmark construction: We collected buggy models from *Stack Overflow* posts and *GitHub* commits to construct the benchmark. In *Stack Overflow*, we select the posts that have a score ≥ 5 and contains the buggy *Keras* code. We used keywords "error", "bug" and synonyms to search for posts. In the second step, we manually reviewed the retrieved posts and removed all the posts that have partial code. In the third step, we applied a second filter by checking if the post provided the training data or used the existing known training data. In the last step, we studied all the answers corresponding to the post ids in the *Stack Overflow*, using the methodology in a prior work [10] that studied the bug fix patterns for deep learning model. We take into consideration the acceptable quality metric and choose the answer that has the highest score. We analyzed each Q&A and derived for each post the fault location and a patch to fix the bug. In total, we obtained 30 posts.

We also mine the *GitHub* commits to collect buggy models. The process consists of three steps. First, we search for all *Keras* repository. After that, we mine all the commits whose title contains keywords used in the *Stack Overflow* mining process described above. Then, we take only the last version fix, and manually check if the commits are related to structure bug or not, from these commits, we derived the fault location and a patch to fix the bug. In the last step, we checked if the repository provided the training data. As a result, we randomly select 11 executable programs with training dataset. For instance, Figure 4 shows the patch we derived for the *Stack Overflow* post # [22]. Table IV and V present our benchmark together with total number of parameters, the line of code, its loss and accuracy before and after fix. Our tool and benchmarks are available for download [21].

Table III shows a comparison between the original *Keras* code and our imperative code in terms of accuracy, losses and runtime. Over all benchmarks, our imperative code has comparable accuracies and losseses compared to *Keras*. The small deviation is due to additional optimizations applied in *Keras* code. In terms of runtime, our imperative code takes around 5 times more execution time compared to *Keras* during training and testing phases. The reason behind that is the extra work done by our imperative code to identify bugs and their root causes. Our code is also able to terminate early when a bug is found and later in our results, we show that we can detect bugs faster than *Keras* callback methods.

All the experiments are run on a computer with Intel(R) Core (TM) i7-6500U, 2.5 GHz processor, and 16 GB of RAM running the 64-bit Windows 8.1 operating system.

B. Results and Analysis

In Table VI and VII, the first column reports the *Stack Overflow* post # with the corresponding link, and *GitHub* repository reference respectively. To compare our results with the results generated from the *Keras* methods, we listed the columns of Time, Epoch, Iteration, IB, and FL, representing how long the approach takes to find the bug, at which epoch and at which iteration the bug is reported, whether the

```
1   1    model = Sequential()
2   2    layer1 = Dense(30,
    3  +              activation='sigmoid',
3   4              input_shape=input_shape,
4   5              kernel_initializer=RandomNormal(stddev=1),
5   6              bias_initializer=RandomNormal(stddev=1))
6   7    model.add(layer1)
7   8    layer2 = Dense(10,
    9  +              activation='softmax',
8  10              kernel_initializer=RandomNormal(stddev=1),
9  11              bias_initializer=RandomNormal(stddev=1))
0  12    model.add(layer2)
1  13
2  14    model.compile(optimizer=SGD(lr=3.0),
3   -              loss='mean_squared_error',
   15  +              loss='categorical_crossentropy',
4  16              metrics=['accuracy'])
```

Fig. 4: (Patch/Fix) the changing to fix a DNN bug [22]

TABLE IV: The Benchmark, we show (Post #) post id from *Stack Overflow*, parameters #, line of code, (Result Before Fix) the loss and accuracy before applying patch, and (Result After Fix) the loss and accuracy value after applying patch

Post #	Total params	LOC	Result Before Fix		Result After Fix	
			Loss	Accuracy	Loss	Accuracy
48385830	23,860	60	NaN	0.10	0.15	0.96
44164749	6,931,610	49	0.38	0.89	0.20	0.92
31556268	17	39	0.25	0.50	0.00	1.00
50306988	12	45	0.70	0.50	0.67	1.00
48251943	17	30	0.37	–	0.63	–
38648195	903	36	0.22	0.41	1.3	0.65
33969059	23	63	72181941.16	–	9766327.81	–
55328966	655,200	52	2.30	0.10	0.85	0.80
34311586	19	29	0.67	–	0.17	–
31880720	9,051	46	7.67	0.50	0.00	1.00
39525358	91	34	0.67	0.62	0.31	1.00
39217567	216	43	0.25	0.50	0.08	0.97
48934338	1,661	37	1335.72	–	341.09	–
47724077	359	46	29.97	0.69	0.55	0.76
59325381	221,226	38	11023.33	0.10	0.03	0.99
59278771	35	43	1.10	0.35	0.39	0.83
52800582	2,701	41	11112.00	–	0.00	–
41372874	15	39	0.15	1.00	0.15	1.00
34673164	5,378	50	0.13	0.78	0.43	0.89
48221692	16	28	2311.60	–	121.94	–
50079585	1,212,513	75	9297.00	0.33	0.94	0.55
45337371	3,221	31	7.85	–	0.69	–
44066044	313	38	9.55	0.38	0.01	1.00
51930566	35	52	0.77	0.64	0.30	0.91
47352366	3,274,634	50	NaN	0.06	0.10	0.98
45442843	4	41	0.50	0.50	0.65	0.83
48594888	4,873,738	60	1.45	0.48	0.86	0.70
31627380	6,658	64	1.24	0.58	0.83	0.63
58609115	9,798	43	44539.79	0.10	0.07	0.99
50481178	4,241	43	0.02	0.99	0.02	1.00

TABLE V: The Benchmark, we show (GH | Ref) the *GitHub* repository reference the number of parameters, line of code (LOC), (Result Before Fix) the loss and accuracy value before applying patch, and (Result After Fix) the loss and accuracy after applying patch

GH	Ref	Total params	LOC	Result Before Fix		Result After Fix	
				Loss	Accuracy	Loss	Accuracy
1	[29]	18989	79	0.39	0.90	0.38	0.90
2	[30]	367107	39	0.82	0.76	0.83	0.75
3	[31]	468980	72	0.72	0.16	0.00	0.75
4	[32]	34438	107	0.65	0.61	0.80	0.71
5	[33]	169431	73	0.30	0.91	0.46	0.86
6	[34]	221	16	0.48	0.78	0.29	0.88
7	[35]	1941	915	44.85	–	44.75	–
8	[36]	341	183	0.02	0.99	0.00	1.00
9	[37]	160305	40	0.28	0.92	0.18	0.94
10	[38]	110627418	67	0.65	0.63	0.43	0.79
11	[39]	9	30	1E-05	–	1E-05	–

approach identifies the bug correctly, and whether the approach successfully reports the fault location of the bug, respectively. Under *IB* and *FL*, ✓ means that the approach is successful, X means it fails, and – is not able to identify or fault the bug.

The results show that our tool is able to identify bugs for 23 out of 29 buggy programs, and our callback function is able to identify bugs for 34 out of 40 buggy programs. In contrast, `TerminateOnNaN()`, `EarlyStopping('loss')`, `EarlyStopping('accuracy')`, and the three *Keras* methods together are able to identify 2, 24, 28, and 32 out of 40 bugs respectively. This is mainly because our technique used a wider spectrum of important runtime values for analysis compared to the other three methods, which only used one metric such as the loss, or accuracy. Also, `EarlyStopping('loss')` and `EarlyStopping('accuracy')` sometimes do not give a good indication that the model has a problem. For example, the deep neural network may be stuck at a local minimum at some point that has values that are close to the global minimum [40].

We failed to detect bugs for 7 out of 30 models using our tool. Our tool is not able to identify the buggy model that predicts a wrong label. In the program [41], the training data are in the range $[-\infty, +\infty]$. This model has an issue that all the negative values are predicted to zeros, because the developer used the ReLU activation function in the last layer instead of activation functions that produce output in the same range of output labels such as TanH.

In the program [42], `fit_generator()` instead of `fit()` is used, which our technique has not yet supported. `fit_generator()` is used to train the model on data generated batch-by-batch [28]. This API is not yet supported by our technique, we plan to cover and investigate other APIs in our future work.

In the *Stack Overflow* post [43], the user asked about the differences between two models, each one has different input dimensions, both models are training correctly, but the differences are in the performance. In this case, our tool will not find any numerical error or misbehavior to detect the bug.

In our tool, we did not target problems such as: (1) lack of dataset, (2) training dataset with distribution problems and (3) incorrect number of epochs, batch sizes, the number of hidden layers, and neural nodes in the layers. An example of these problems can be found in programs [44], [45], these problems make the model terminates training at an early stage, and our tool will not detect the issue in the model.

We also compared our approach against three *Keras* methods in terms of fault localization, We found that none of the three *Keras* approaches are able to determine the root causes. Our tool is able to determine root causes for 19 out 29 programs, and our callback is able to determine root causes for 21 out of 40 programs. We compared the results from our tool and our callback method with the ground truth we built for the benchmark. Our tool can precisely determine the layer or the parameter that causes the error for 19 out 29 cases and our callback method can precisely determine for 21 out 40 cases. In the rest of the models, the tool and the callback

method reported the root cause in another layer because the training process is a cycle; the operation in one layer can affect the adjacent layer. For example, the program [46], `mean_absolute_error` is used as a loss function instead of `mean_squared_error`, and the `SGD` optimizer is used instead of `Adam` optimizer. Our tool reports that the bug is EBW (error in weight in backward propagation) at layer. No = 1. In the program [47], the user assigned the learning rate = 0.1 instead of learning rate = 0.001. Our tool reported EBA (Error before activation function), layer. No = 2.
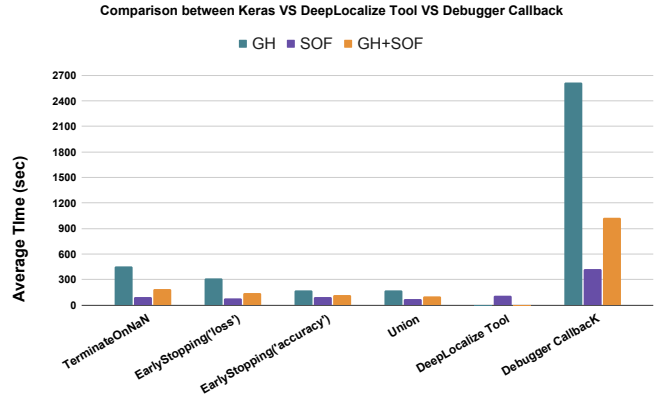
Fig. 5: Comparison between *Keras* and Our Technique

We measured the analysis time for the three *Keras* methods, our tool, and our callback function using *Stack Overflow* (SOF) and *GitHub* (GH) benchmarks. Using SOF benchmarks, `TerminateOnNaN()`, `EarlyStopping('loss')`, `EarlyStopping('accuracy')`, the three *Keras* methods together, our callback method and our tool take an average of 93.01, 78.98, 93.04, 73.55, 421.39, and 107.99 seconds respectively overall all *Stack Overflow* (SOF) benchmarks. While in GH benchmarks, `TerminateOnNaN()`, `EarlyStopping('loss')`, `EarlyStopping('accuracy')`, the three *Keras* methods together, and our callback take an average of 451.52, 310.57, 171.22, 172.02, and 2613.6 seconds respectively. The average time for all *Stack Overflow* (SOF) and *GitHub* (GH) benchmarks is 191.6, 142.67, 114.54, 100.63, and 1024.25 seconds respectively for the four corresponding tools. In Figure 5, we plot the average time, and shown that, `TerminateOnNaN()` terminates the program when there is a NaN in the loss value; in other cases, the training will continue. This is why it takes longer than other methods. Our callback function needs more value than the three *Keras* methods, such as update weight and gradient to perform dynamic analysis. This leads to an overhead if the model has a large number of parameters, but the overhead is usually insignificant compared to the amount of time that takes other callback functions. However, we anticipate that the time needed to manually debug and fault localization would be much more than the time required to execute our callback function.

During building the benchmark from *Stack Overflow* and

GitHub and identifying the bugs using our technique, we observed that the most frequent bugs that the developer made are (1) choosing the activation, and loss function, as there are many combinations to choose the activation and loss depend on the type of the problem, (2) selecting the hyperparameters, such as the learning rate, the number of neural nodes in the layer, number of layers in the model and (3) preprocessing the training dataset, for example, the MNIST model needs to normalize the training dataset from the range [-256, + 256] to [-1, 1]. Without fault localization information, the developers will face the problem of determining the layers or parameters that induce the bug in the model.

C. Summary

1) RQ1 (Validation): We provide the empirical evidence (Table III) that training a DNN using our imperative program is consistent with using the *Keras* library. For RQ1, we show that our tool is able to identify 23 out of 29 faulty models, and our callback function is able to identify 34 out of 40 faulty models. Also our technique is able to determine the fault localization precisely for 19 out 29 using our tool, and 21 out 40 using our callback while *Keras* is not able to localize bug.

2) RQ2 (Comparison): Our technique is able to identify 34 out of 40 faulty models using our callback functions, and 23 out of 29 using our tool. On the other hand, the *Keras* library reported bugs for 2, 24, and 28 out of 40 models, respectively. Also, if we take the union of three methods, the *Keras* library reported bugs for 32 out of 40 models. We also show that our technique can effectively locate the root causes for 21 out of 40 programs using our callback and 19 out 29 using our tool while *Keras* methods does not support this feature. As it can be seen in Figure 5, our tool using dynamic analysis is practical and faster than *Keras* by identifying the bug. Three methods using *Keras* take on average 191.6, 142.67, and 114.54 seconds respectively, while our tool takes on average 107.99 seconds. On the other hand, our callback needs more time to make dynamic analysis for additional parameters than loss and accuracy. This overhead is usually insignificant compared to the amount of time taken by `TerminateOnNaN()`.

3) RQ3 (Limitation): We failed to detect bugs for 7 out of 30 programs and failed to localize faults for 11 out of 30 programs using our tool, and our callback failed to localize fault for 20 out of 41. These programs either predicted wrong labels or have problems in the training dataset, the settings of epoches, batch sizes, the number hidden layers, and the number of neural nodes. In such cases, the training stopped before the model finished learning. Thus our tool was not able to handle these cases. Also, our prototype does not support all the *Keras* APIs. As an example, one of the buggy program in *Stack Overflow* [42] used `fit_generator()` instead of `fit()` function. `fit_generator()` is used to train the model on data generated batch-by-batch [28]. We plan to cover and investigate other APIs in our future work.

V. THREATS TO VALIDITY

We were mainly concerned with the implementation of our tool that may affect the evaluation and result. To minimize this threat, we followed *Keras* code [70] and carefully reviewed our implementation between the authors to reduce the chances that major errors were compared to our tool against *Keras*, as we have shown in Table III.

Our tool currently handles the frequent layers/APIs in the *Keras* library, including: `Dense`, `Dropout`, `MaxPooling2D`, `Conv2D`, and `BatchNormalization`. We may not yet able to handle deep learning programs whose bugs are related to other APIs e.g., `ConvLSTM2D()`, `Conv3D()`.

We validated the output of our tool using the benchmarks we created. The trustworthy of manually created benchmarks could be a threat to the validity of our results. To alleviate this threat, we follow the same methodology in prior works [13]. Also, we have evaluated the benchmark before/after fixing the issue in the model.

VI. RELATED WORK

The closest related work in terms of technical ideas is by Gopinath *et al.* [20], [24]. Gopinath *et al.* proposed a new approach (DeepCheck) inspired from program analysis to test a Deep Neural Network (DNN) using symbolic execution. DeepCheck also uses a white box technique to enable symbolic execution to find the important pixels, and find the attack pixels by translating DNN to an imperative program that has the same behavior as DNN. The experimental results conducted using MNIST data set shows that their approach is able to create 1-pixel and 2-pixel attacks by finding the most important pixels that DNN fails to classify correctly. Our imperative representation is inspired by this work. While this work focuses on identifying adversarial attack, our work aims to identify faults in DNNs and localize their fault.

A. Testing Deep Neural Networks

DeepTest [71] and follow-up work aim to automatically generate test cases to examine corner cases corresponding to real world inputs. These works have focused on test case generation, whereas our work uses an existing training dataset and focuses on localizing the root cause of a bug by observing the runtime behavior of a deep neural network. Zhang *et al.* [72] present a comprehensive survey of work in this area. Eniser *et al.* [73] proposed a new white box analysis technique (DeepFault) inspired from spectrum-based fault localization. DeepFault tests DNN models to achieve two objectives: (i) detect suspicious neurons, i.e., neurons likely to be more responsible for inadequate DNN performance; and (ii) synthesis of new inputs, using correctly classified inputs, that exercise the identified suspicious neurons. The experimental results conducted on MNIST and CIFAR-10 datasets show that DeepFault is effectively identifying suspicious neurons. DeepFault does not focus on structure bugs, instead they focus on training bug, and pre-trained DNNs analysis, given a specific test set. While our tool identifies the buggy model and faults the root causes of structure bug in DNN.

TABLE VI: Comparisons Between the *Keras* Methods and Our Tool, and Our Callback Using *Stack Overflow* Post

Post #	Ref	TerminateOnNaN					EarlyStopping(monitor='loss')					EarlyStopping(monitor='accuracy')					Our Tool					Our CallBack				
		Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL
48385830	[22]	1.20	1	10	✓	–	12.21	1	60000	✓	–	34.90	3	60000	✓	–	0.14	1	4	✓	✓	2.14	1	20	✓	✓
44164749	[48]	175.23	5	60000	X	–	175.23	5	60000	X	–	175.23	5	60000	X	–	22.93	1	50	✓	✓	111.56	1	2000	✓	✓
31556268	[46]	5.69	1000	4	X	–	5.69	1000	4	X	–	0.91	2	4	✓	–	4.57	98	394	✓	X	1.20	1	4	✓	X
50306988	[49]	1.44	5	200	X	–	1.44	5	200	X	–	1.30	2	200	✓	–	1.03	1	100	✓	✓	3.57	1	64	✓	✓
48251943	[41]	5.22	100	17	X	–	0.93	27	17	✓	–	5.22	100	17	X	–	20.45	100	17	X	X	706.83	100	17	X	X
38648195	[50]	120.52	20	48000	X	–	120.52	20	48000	X	–	120.00	20	48000	X	–	2.49	1	95	✓	X	25.92	1	100	✓	✓
33969059	[51]	130.66	20	10000	X	–	14.03	2	10000	✓	–	14.14	2	10000	✓	–	0.95	1	50	✓	✓	1.52	1	32	✓	✓
55328966	[52]	98.84	10	49999	X	–	98.84	10	49999	X	–	98.84	10	49999	X	–	272.70	1	737	✓	✓	9092.93	10	49999	X	X
34311586	[53]	1.01	20	6	X	–	0.87	2	6	✓	–	0.97	20	6	X	–	0.73	8	50	✓	✓	1.02	1	32	✓	✓
31880720	[54]	176.02	3	20000	X	–	110.25	2	20000	✓	–	113.28	2	20000	✓	–	1.11	1	50	✓	✓	2.12	1	1	✓	✓
39525358	[44]	3.15	150	13	X	–	3.15	150	13	X	–	1.68	2	13	✓	–	28.36	150	13	X	X	14.29	2	10	✓	✓
39217567	[55]	16.97	1000	256	X	–	1.31	11	256	X	–	1.18	2	256	✓	–	329.46	100	256	X	X	188.71	5	50	✓	X
48934338	[47]	11.03	1000	50	X	–	0.91	10	50	✓	–	0.85	2	50	✓	–	0.33	1	13	✓	✓	3.01	1	32	✓	X
47724077	[56]	31.07	50	32561	X	–	2.92	3	32561	✓	–	3.46	4	32561	✓	–	0.66	1	50	✓	✓	1.42	1	100	✓	✓
59325381	[57]	758.04	10	60000	X	–	260.06	3	60000	✓	–	239.52	3	60000	✓	–	125.52	1	1750	✓	X	24.66	1	160	✓	✓
59278771	[58]	1.94	200	135	X	–	5.24	62	135	✓	–	1.94	200	135	X	–	0.52	1	50	✓	✓	60.64	1	75	✓	X
52800582	[59]	10.04	1000	400	X	–	1.79	2	400	✓	–	10.04	1000	400	X	–	0.14	1	7	✓	✓	12.79	4	400	✓	X
41372874	[60]	6.86	20	239	X	–	6.86	20	239	X	–	2.40	3	239	✓	–	66.65	20	239	X	X	1.42	1	2	✓	✓
34673164	[61]	1.93	9	20	X	–	2.00	2	20	✓	–	2.09	6	20	✓	–	2.04	11	100	✓	✓	26.00	5	9	✓	✓
48221692	[62]	7.16	1000	150	X	–	2.61	280	150	✓	–	0.95	2	150	✓	–	24.24	13	150	✓	✓	8.02	4	150	✓	X
50079585	[42]	171.69	10	61	X	–	172.00	10	61	X	–	54.38	3	61	✓	–	–	–	–	–	–	–	–	–	–	–
45337371	[63]	12.52	5	1000	X	–	6.79	2	1000	✓	–	7.02	2	1000	✓	–	1.17	1	50	✓	✓	2.43	1	1	✓	✓
44066044	[64]	3.87	100	21	X	–	2.07	4	21	✓	–	1.97	2	21	✓	–	1.07	2	50	✓	✓	1.41	1	5	✓	✓
51930566	[43]	4.07	50	75	X	–	4.07	50	75	X	–	1.66	2	75	✓	–	38.65	50	75	X	X	22.40	1	40	✓	X
47352366	[65]	32.75	1	4950	✓	–	381.31	1	60000	✓	–	793.64	2	60000	✓	–	75.46	1	50	✓	✓	1177.03	1	1050	✓	X
45442843	[66]	2.57	100	200	X	–	1.17	3	200	✓	–	1.15	2	200	✓	–	0.48	1	50	✓	✓	0.97	1	200	✓	✓
48594888	[67]	1055.29	5	50000	X	–	1055.29	5	50000	X	–	1055.29	5	50000	X	–	116.38	1	51	✓	✓	4.24	1	32	✓	✓
31627380	[68]	6.28	10	712	X	–	6.12	10	712	X	–	3.09	3	712	✓	–	1.28	1	50	✓	✓	1.76	1	4	✓	✓
58609115	[69]	3.12	10	1767	X	–	2.61	8	1767	✓	–	1.94	3	1767	✓	–	29.06	1	1350	✓	✓	343.78	1	1250	✓	X
50481178	[45]	12.66	50	200	X	–	4.12	6	200	✓	–	3.58	3	200	✓	–	1963.13	50	200	X	X	376.44	1	150	✓	X

TABLE VII: Comparisons Between the *Keras* Methods and Our Callback Using *GitHub* Repository

GH	Ref	TerminateOnNaN					EarlyStopping(monitor='loss')					EarlyStopping(monitor='accuracy')					Union					Our CallBack				
		Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL	Time	Epoch	Iteration	IB	FL
1	[29]	356.14	2	1944601	X	–	345.82	2	1944601	X	–	351.46	2	1944601	X	–	343.17	2	1944601	X	–	11.80	1	96	✓	X
2	[30]	3427.71	50	100	X	–	2602.44	36	100	✓	–	1082.78	15	100	✓	–	1073.39	15	100	✓	–	8432.06	50	100	X	X
3	[31]	101.78	100	100	X	–	9.01	8	2140	✓	–	4.39	5	2140	✓	–	7.20	5	2140	✓	–	31.69	1	96	✓	✓
4	[32]	256.76	20	1999	X	–	52.46	4	1999	✓	–	54.10	4	1999	✓	–	52.88	20	1999	✓	–	102.44	1	300	✓	X
5	[33]	168.86	10	27839	X	–	169.24	10	27839	X	–	171.04	10	27839	X	–	191.71	10	27839	X	–	164.70	1	2560	✓	✓
6	[34]	19.92	25	768	X	–	2.44	10	768	✓	–	1.54	2	768	✓	–	1.52	2	768	✓	–	9568.09	25	768	X	X
7	[35]	19.33	10	1166	X	–	19.75	10	1166	X	–	4.90	3	1166	✓	–	4.95	3	1166	✓	–	1.90	2	1	✓	✓
8	[36]	2.66	2	125	X	–	2.51	2	125	X	–	2.74	2	125	✓	–	2.58	2	125	✓	–	1022.32	2	125	X	X
9	[37]	13.30	4	15000	X	–	12.67	4	15000	X	–	13.55	4	15000	X	–	13.38	4	15000	X	–	9381.21	4	15000	X	X
10	[38]	587.26	15	1440	X	–	194.23	5	1440	✓	–	193.74	5	1440	✓	–	198.24	5	1440	✓	–	28.04	1	64	✓	✓
11	[39]	12.97	10	10000	X	–	5.73	4	10000	✓	–	3.20	2	10000	✓	–	3.23	2	10000	✓	–	5.38	1	30	✓	X

B. Empirical Study on Deep Learning Bugs

There have been several empirical studies that have analyzed different kind of bugs in deep learning networks. These studies have been conducted on real code and examples from the *Stack Overflow* posts and *GitHub* issues. They have focused on symptoms and root causes of bugs to have a better understanding of deep learning bugs.

Zhang *et al.* [8] utilized *Stack Overflow* posts and *GitHub* commits to investigate bugs in deep learning applications built on top of TensorFlow. They focused on symptoms and root causes of TensorFlow bugs to have a better understanding of deep learning bugs. Islam *et al.* [9] also studied deep learning bugs using *Stack Overflow* questions and *GitHub* commits. They also adapted a taxonomy of bug type, root cause, and bug impacts for deep learning software for five popular deep learning libraries.

Another study has been conducted to understand the bug fix pattern and how the developer can develop a tool to fix the bugs automatically by Islam *et al.* [10]. They have conducted a comprehensive study on 415 posts from *Stack Overflow*, and 555 from *GitHub* commits for five popular DNN packages to understand the bug fix patterns, and how the bugs can be fixed in DNN software. The main goal of this study to help the developers to understand the characteristics of bug and how they can design an automated repair tool.

C. Bugs Repairing in Deep Learning

In recent years, there are several proposals for debugging deep neural networks. These techniques are often inspired from software debugging and testing techniques.

Ma *et al.* [74] proposed and developed a technique called MODE inspired by software debugging. MODE performs state differential analysis to solve two types of problems: overfitting problems and under-fitting problems. MODE can solve these problems by identify the buggy features (or neurons) that are responsible for the misclassification in the model, then it constructs the degree of importance of features to retrain the faulty neurons with new input samples selection. MODE provides effective and efficient method to fix the buggy models without introducing new bugs, and compromise on accuracy and training time cost.

Zhang *et al.* [75] introduced an automatic approach to fixing deep learning models called Apricot. Apricot is able to adjust the ill-trained weights without using additional training data or any artificial parameters, Apricot using a set of reduced models from the original model, and compare the differences between the original model and correct/incorrect of reduced models iteratively, to find these failing test case that are responsible for the misclassification in the original model. The approach uses three strategies to adjust the weight and achieve a higher test accuracy. The experimental results using CIFAR-10 dataset and five state-of-the-art of deep learning models have shown that the approach can increase the test and training accuracy.

In recent years, several researchers are supporting automated debugging and repair approaches for deep neural networks, and recent research is summarized in [72]. This topic is still at the early stages [74], [75]. To the best of our knowledge, all previous works are focused on the training bugs. Our technique

is the first approach that automatically identifies the buggy model and localizes the root causes of structure bug in DNNs by applying the DNN Bug Detection algorithm.

VII. Conclusions and Future Work

As deep neural networks are becoming integrated with software systems from different domains, developing debugging techniques to localize the root cause of the bug has become urgent. Thus, recent work has developed techniques to inspect the entire model and find faults. Inspired by program analysis and software debugging techniques, we have presented an automated approach powered by a dynamic analysis and statistical analysis. It can help identify the buggy model and the root causes of DNN error. An experimental evaluation using 40 real deep learning applications shows the usefulness of our technique. For 34/40 cases, our approach was able to detect faults whereas the best debugging approach provided by *Keras* detected 32/40 faults. Our approach was able to localize 21/40 bugs whereas *Keras* did not localize any faults.

Future work includes developing techniques to repair deep neural network bugs, and exploring cases that our work was unable to detect faults (6/40) and localize errors (19/40). Recent work has also used analysis of the DNN structure to decompose it into modules [76]. It would be interesting to explore whether a similar mechanism can be utilized for better localization. It would also be interesting to go beyond accuracy bugs to detect and localize more non-functional bugs, e.g. fairness bugs [77].

VIII. Acknowledgment

References

[1] Martin Abadi et al, "TensorFlow: large-scale machine learning on heterogeneous systems," 2015, https://www.tensorflow.org/.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[3] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.

[5] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: a Python framework for fast computation of mathematical expressions," *arXiv e-prints*, pp. arXiv–1605, 2016.

[6] Francois Chollet, "Keras: the Python deep learning library," 2015, https://keras.io/.

[7] T. Zhang, C. Gao, L. Ma, M. R. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *The 30th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2019.

[8] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[9] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[10] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: fix patterns and challenges," in *ICSE'20: The 42nd International Conference on Software Engineering*, May 23-May 29, 2020 2020.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: a generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[12] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[13] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.

[14] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 82–91.

[15] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 253–267, 1999.

[16] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 272–281.

[17] I. Vessey, "Expertise in debugging computer programs: a process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[18] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. IEEE, 2003, pp. 30–39.

[19] A. Zeller, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[20] D. Gopinath, M. Zhang, K. Wang, I. B. Kadron, C. S. Pasareanu, and S. Khurshid, "Symbolic execution for importance analysis and adversarial generation in neural networks," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE*, 2019.

[21] https://github.com/Wardat-ISU/DeepLocalize, 2021, [Online; accessed 12-Feb-2021].

[22] "Simple Keras neural network isn't learning," 2018, https://stackoverflow.com/questions/48385830/.

[23] Francois Chollet, "Usage of callbacks," 2015, https://keras.io/callbacks/.

[24] D. Gopinath, C. S. Pasareanu, K. Wang, M. Zhang, and S. Khurshid, "Symbolic execution for attribution and attack synthesis in neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 282–283.

[25] K. C. Sim, A. Narayanan, T. Bagby, T. N. Sainath, and M. Bacchiani, "Improving the efficiency of forward-backward algorithm using batched computation in TensorFlow," in *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 2017, pp. 258–264.

[26] J. A. Hertz, *Introduction to the theory of neural computation*. CRC Press, 2018.

[27] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[28] Francois Chollet, "Keras documentation," 2015, https://keras.io/.

[29] https://github.com/Jacobvs/DDOS-ML-Detection, 2017, [Online; accessed 20-Aug-2020].

[30] https://github.com/kcct-fujimotolab/keras-super-resolution, 2017, [Online; accessed 20-Aug-2020].

[31] https://github.com/ShyamPoovaiah/FacialKeypointsDetection, 2018, [Online; accessed 20-Aug-2020].

[32] https://github.com/ronnyworm/keras_catsdogs, 2017, [Online; accessed 20-Aug-2020].

[33] https://github.com/ulmefors/Traffic-Sign-Classifier, 2017, [Online; accessed 20-Aug-2020].

[34] https://github.com/eangelou/kerasma, 2017, [Online; accessed 20-Aug-2020].

[35] https://github.com/yetanotherdeveloper/kaggle-melbourne_properties, 2018, [Online; accessed 20-Aug-2020].

[36] https://github.com/alexander-travov/gol_nn, 2019, [Online; accessed 20-Aug-2020].

[37] https://github.com/tautvydasversockas/MovieReviews_Keras, 2019, [Online; accessed 20-Aug-2020].

[38] https://github.com/sammyhaq/hotdog-sandwich, 2019, [Online; accessed 20-Aug-2020].

[39] https://github.com/vincenty2022/Averaging_Neural_Networks, 2016, [Online; accessed 20-Aug-2020].

[40] K. Kawaguchi, "Deep learning without poor local minima," in *Advances in neural information processing systems*, 2016, pp. 586–594.

[41] "my Keras model does not predict negative values," https://stackoverflow.com/questions/48251943/, 2018, [Online; accessed 19-Feb-2020].

[42] "CNN with keras, accuracy not improving," https://stackoverflow.com/questions/50079585/, 2019, [Online; accessed 19-Feb-2020].

[43] "Input nodes in Keras NN," https://stackoverflow.com/questions/51930566/, 2019, [Online; accessed 19-Feb-2020].

[44] "Neural network accuracy optimization," https://stackoverflow.com/questions/39525358/, 2017, [Online; accessed 19-Feb-2020].

[45] "keras MLP accuracy zero," https://stackoverflow.com/questions/50481178/, 2019, [Online; accessed 19-Feb-2020].

[46] "How to use keras for XOR," https://stackoverflow.com/questions/31556268/, 2016, [Online; accessed 19-Feb-2020].

[47] "Non linear Regression: Why isn't the model learning?" https://stackoverflow.com/questions/48934338/, 2018, [Online; accessed 19-Feb-2020].

[48] "How does Keras handle multilabel classification?" https://stackoverflow.com/questions/44164749/, 2018, [Online; accessed 19-Feb-2020].

[49] "Neural net fails on toy dataset," https://stackoverflow.com/questions/50306988/, 2019, [Online; accessed 19-Feb-2020].

[50] "Keras low accuracy classification task," https://stackoverflow.com/questions/38648195/, 2017, [Online; accessed 19-Feb-2020].

[51] "Trying to get simple Keras neural net example to work," https://stackoverflow.com/questions/33969059/, 2016, [Online; accessed 19-Feb-2020].

[52] "tf.keras loss becomes NaN," https://stackoverflow.com/questions/55328966/, 2019, [Online; accessed 19-Feb-2020].

[53] "Cannot train a neural network solving XOR mapping," https://stackoverflow.com/questions/34311586/, 2016, [Online; accessed 19-Feb-2020].

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

[55] "Keras neural network outputs same result for every input," https://stackoverflow.com/questions/39217567/, 2017, [Online; accessed 19-Feb-2020].

[56] "Keras: could not broadcast input array from shape (14,1) into shape (14)," https://stackoverflow.com/questions/47724077/, 2018, [Online; accessed 19-Feb-2020].

[57] "Low accuracy after training a CNN," https://stackoverflow.com/questions/59325381/, 2018, [Online; accessed 19-Feb-2020].

[58] "Super low accuracy for neural network model," https://stackoverflow.com/questions/59278771/, 2017, [Online; accessed 19-Feb-2020].

[59] "Why do I fail to predict y=x**4 with Keras? (y=x**3 works)," https://stackoverflow.com/questions/52800582/, 2019, [Online; accessed 19-Feb-2020].

[60] "Binary classification with Keras: poor performance," https://stackoverflow.com/questions/41372874/, 2017, [Online; accessed 19-Feb-2020].

[61] "How to train and tune an artificial multilayer perceptron neural network using Keras?" https://stackoverflow.com/questions/34673164/, 2016, [Online; accessed 19-Feb-2020].

[62] "Create a square function estimator with Keras," https://stackoverflow.com/questions/48221692/, 2018, [Online; accessed 19-Feb-2020].

[63] "Taking derivative of Keras model wrt to inputs is returning all zeros," https://stackoverflow.com/questions/45337371/, 2018, [Online; accessed 19-Feb-2020].

[64] "Keras accuracy is not increasing over 50%," https://stackoverflow.com/questions/44066044/, 2018, [Online; accessed 19-Feb-2020].

[65] "Keras unreasonnably slower than TensorFlow," https://stackoverflow.com/questions/47352366/, 2018, [Online; accessed 19-Feb-2020].

[66] "Sigmoid layer in Keras," https://stackoverflow.com/questions/45442843/, 2018, [Online; accessed 19-Feb-2020].

[67] "CNN train accuracy gets better during training, but test accuracy stays around 40%," https://stackoverflow.com/questions/48594888/, 2018, [Online; accessed 19-Feb-2020].

[68] "Trying Kaggle Titanic with keras," https://stackoverflow.com/questions/31627380/, 2016, [Online; accessed 19-Feb-2020].

[69] "Manual predictions of neural net go wrong," https://stackoverflow.com/questions/58609115/, 2016, [Online; accessed 19-Feb-2020].

[70] Francois Chollet, "Keras: deep learning for humans," 2015, https://github.com/keras-team/keras.

[71] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–314. [Online]. Available: https://doi.org/10.1145/3180155.3180220

[72] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[73] H. F. Eniser, S. Gerasimou, and A. Sen, "DeepFault: fault localization for deep neural networks," in *Fundamental Approaches to Software Engineering*, R. Hähnle and W. van der Aalst, Eds. Cham: Springer International Publishing, 2019, pp. 171–191.

[74] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 175–186.

[75] H. Zhang and W. Chan, "Apricot: a weight-adaptation approach to fixing deep learning models," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 376–387.

[76] R. Pan and H. Rajan, "On decomposing a deep neural network into modules," in *ESEC/FSE'2020: The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 8-November 13, 2020 2020.

[77] S. Biswas and H. Rajan, "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*, November 8-November 13, 2020 2020.