Repairing Deep Neural Networks: Fix Patterns and Challenges

Md Johirul Islam
mislam@iastate.edu
Iowa State University
Ames, IA, USA

Rangeet Pan
rangeet@iastate.edu
Iowa State University
Ames, IA, USA

Giang Nguyen
gnguyen@iastate.edu
Iowa State University
Ames, IA, USA

Hridesh Rajan
hridesh@iastate.edu
Iowa State University
Ames, IA, USA

ABSTRACT

Significant interest in applying Deep Neural Network (DNN) has fueled the need to support engineering of software that uses DNNs. Repairing software that uses DNNs is one such unmistakable SE need where automated tools could be very helpful; however, we do not fully understand challenges to repairing and patterns that are utilized when manually repairing them. What challenges should automated repair tools address? What are the repair patterns whose automation could help developers? Which repair patterns should be assigned a higher priority for automation? This work presents a comprehensive study of bug fix patterns to address these questions. We have studied 415 repairs from *Stack Overflow* and 555 repairs from *GitHub* for five popular deep learning libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* to understand challenges in repairs and bug repair patterns. Our key findings reveal that DNN bug fix patterns are distinctive compared to traditional bug fix patterns; the most common bug fix patterns are fixing data dimension and neural network connectivity; DNN bug fixes have the potential to introduce adversarial vulnerabilities; DNN bug fixes frequently introduce new bugs; and DNN bug localization, reuse of trained model, and coping with frequent releases are major challenges faced by developers when fixing bugs. We also contribute a benchmark of 667 DNN (bug, repair) instances.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

deep neural networks, bugs, bug fix, bug fix patterns

1 INTRODUCTION

The availability of big data has fueled the emergence of deep neural networks (DNN). A DNN consists of a set of layers. Each layer contains a set of nodes collecting inputs from the previous layer and feeding output to nodes in the next layer via a set of weighted edges. These weights are adjusted using examples, called training data, and set to values that minimize the difference between actual outputs of the DNN and expected outputs measured using an objective function called loss function. The availability of big data has made it possible to adjust accurately weights for DNNs containing many layers. Thus, many software systems are routinely utilizing DNNs. SE for DNNs has thus become important.

A significant SE problem in the software that uses DNNs is the presence of bugs. What are the common bugs in such software? How do they differ? Answering these questions has the potential to fuel SE research on bug detection and repair for DNNs. Fortunately, recent work has shed some light on this issue. Zhang *et al.* [35] have identified bug types, root causes, and their effects in *Tensorflow* library for DNN. Islam *et al.* [10] have studied an even larger set of libraries including *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch* to identify bug characteristics. While Zhang *et al.*'s work presents an initial study on repair patterns for *Tensorflow*, these works have not focused on the characteristics of repairs. Since repairing software that uses DNNs is an unmistakable SE need where automated tools could be very helpful, fully understanding the challenges to repairing and patterns that are utilized when manually repairing bugs in DNNs is critical. What challenges should automated repair tools address? What are the repair patterns whose automation could help developers? Which repair patterns should be prioritized?

Motivated by these questions, we conduct a comprehensive study of bug repair patterns for five DNN libraries *Caffe*, *Keras*, *Tensorflow*, *Theano*, and *Torch*. We leverage the dataset of DNN bugs published by Islam *et al.* [10] that consists of 415 bugs from *Stack Overflow* and 555 bugs from *GitHub*. We then collect the code snippets used to fix these bugs from both *Stack Overflow* and *GitHub*. We then manually study these repairs and label them according to a classification scheme developed using the open coding approach. To study the

Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan

fix in *Stack Overflow* we study the accepted answers and answers with score >= 5 from *Stack Overflow* post that fixes the bug in the original post. To study the bug fix patterns in *GitHub*, we take the bug-fix commits in the dataset and study the code that is changed to fix the bug. If we do not find any fixes that match our selection criteria in *Stack Overflow* and relevant fix in *GitHub* we discard those bugs. In total, we have studied *320* bug fix codes in *Stack Overflow* and *347* bug fix codes in *GitHub*. We have also analyzed these bug fixes to answer the following research questions:

RQ1 **(Common bug fix patterns)** What are the most common bug fix patterns?

RQ2 **(Fix pattern across bug types)** Are the bug fix patterns different for different bug types?

RQ3 **(Fix pattern across libraries)** Are the bug fix pattern different for different libraries?

RQ4 **(Risk in fix)** Does fixing a DNN bug introduces a new bug?

RQ5 **(Challenges)** What are the challenges in fixing DNN bugs?

Our key findings are as follows: DNN bug fix patterns are distinctive compared to traditional bug fix patterns; the most common bug fix patterns are fixing data dimension and network connectivity; DNN bug fixes have the potential to introduce adversarial vulnerabilities [9]; DNN bug fixes frequently introduce new bugs; and DNN bug localization, reuse of trained model, and coping with frequent releases are major challenges faced by developers when fixing bugs. We also contribute a benchmark of 667 DNN (bug, repair) instances.

2 METHODOLOGY

2.1 Dataset

In our study, we build on the bug dataset prepared by Islam *et al.* [10] to collect and prepare the dataset of bug fixes. The bug dataset contains 415 bugs from *Stack Overflow* and 555 bugs from *GitHub* for 5 different deep learning libraries as shown in Table 1.

Table 1: Summary of the bug repair dataset.

Library	Stack Overflow		GitHub	
	Bugs [10]	Fixes (current)	Bugs [10]	Fixes (current)
Caffe	35	**27**	26	**17**
Keras	162	**143**	348	**167**
Tensorflow	166	**118**	100	**90**
Theano	27	**15**	35	**32**
Torch	25	**17**	46	**41**
Total	415	**320**	555	**347**

Collecting *Stack Overflow* bug fixes: To collect the bug fixes in *Stack Overflow* bug dataset, we study all the answers corresponding to the post ids in *Stack Overflow* bug dataset. If a post has accepted answer with code, then we consider that code snippet as a fix. If the accepted answer doesn't have code but describes what needs to be fixed in original bug we consider those as fix as well. If a bug post does not have an accepted answer but has an answer with >= 5 scores we consider them as fixes also as score 5 is considered as an acceptable quality metric in prior works [10]. Following this methodology, we were able to find 320 fixes for 320 bug related posts in the *Stack Overflow* dataset.

Table 2: Summary of the bug fix patterns.

Bug Fix Pattern	Definition
Loss function	add, remove or replace the loss function.
Neural connection	change node connectivity in the DNN, e.g. change weights, remove edges, add backward propagation.
Add layer	add another layer to the DNN model
Layer dimension	change a layer's input and output size, e.g. to make it compatible with adjacent layers' dimension
Data dimension	align the input data's dimension with the layer dimension
Accuracy metric	replace the accuracy metric being used to measure the correctness of a model, often to match better
Data type	change the type of data given as input to the DNN
Activation	change the activation function used in the DNN
Iterations	change the number of times the training would be done, e.g. modify batch size, epoch or add a loop
Versioning	adapt the code to the new version of the library
API contract	fix API compositions so that the output of an API meets the preconditions of another API
Data wrangling	fix the form of the data for downstream operations without modifying its intent
Monitor	add diagnostics code to monitor training
Optimizer	change the optimization function used by the DNN
Change neural architecture	overhaul the design of the DNN's architecture including a new set of layers and hyperparameters, generally because changes above can't fix the bug

Collecting *GitHub* bug fixes: To collect *GitHub* bug fixes we went to the link of the buggy code snippets in the dataset. If the code snippet was fixed in a later revision, then we take those fixes. A single line may contain multiple bugs [10]. A single bug fix commit might fix multiple bugs. We consider them different fixes. For example, in the same fix API name is updated from deprecated to a new version and the dimension is also fixed. We consider them as two different fixes. Some of the bugs are not yet fixed in the repositories and some repositories have been made private or deleted since the previous study. We omitted those bugs. Following this methodology, we collected 347 bug fixes from *GitHub*.

2.2 Bug Fix Pattern Classification

Next, we created a classification scheme to manually label the bug fix dataset. We started with the classification scheme used by Pan, Kim, and Whitehead [20] and found that their classification scheme has 28 non-ML bug fix categories and among them only 4 fix categories are applicable for the DNN-related fixes. Then, we used the open coding approach to refine it to come with a pattern of 15 different kinds of DNN-specific bug fix patterns. We conducted a pilot study where two Ph.D. students individually studied the fixes to come with possible classification needed. Each student proposed a set of classes that were then reconciled during an in-person meeting where all the authors were present. Our pilot study revealed that there are a number of unique bug fix patterns in our DNN setting. Therefore, the classification from prior work had to be

significantly modified. The final classification is shown in Table 2 and discussed below.

> 💡 **Finding 1** ⟹ We found that DNN bug fix patterns are very different from traditional bug fix patterns such as [20].

2.2.1 Loss Function. This group of fixes is based on the addition, removal or update of the loss function during training. The loss function is a key parameter that helps the training process to identify the deviation from the learned and actual examples. Different kind of problems demands different loss function, e.g. cross-entropy loss is widely used in the classification problems whereas mean square error loss (MSE) is mostly used for regression-based problems. Some problem asks for custom loss function for better training result and we group this kind of fixes into this class.

2.2.2 Neural Connection. This group of fixes changes the connection between nodes in the DNN. A DNN is a graph, where edges are the weights and bias and nodes are the elements of each layer. For example, in a dense layer, the weight edges are fully connected with the next layer and the dimension of the layer determines the number of nodes to be available in that layer. Those bug fixes that reconfigure these connections for better results are classified in this category. The changes include change of weight, removing edges by pruning the network, adding backward propagation , etc.

2.2.3 Add Layer. In any classification based problem, there will be at least two layers in the model, input layer and the output layer. To learn the features of the input, a DNN frequently needs more intermediate layers (called hidden). This group of fixes adds more layers to the DNN to improve performance. Added layers can be dense, where two consecutive layers are fully connected, convolution layer, where convolution function has been applied to the input, dropout layer for reducing the overfitting, etc.

2.2.4 Layer Dimension. These fixes change the dimensions of the layers to make them compatible with adjacent layers and input.

2.2.5 Data Dimension. Data dimension related fix is similar to layer dimension but it is related to the input data rather than to the DNN layers. The dimension of the data needs to be aligned with the DNN. This type of fixes is mostly needed when the input dimensions of the DNN and the data dimension do not match.

2.2.6 Accuracy Metric. To measure the correctness of a DNN, the accuracy metric is one of the key parameters to be configured. Type of the problems has a huge influence on the type of accuracy metric to be used, e.g. classification problems are judged using classification accuracy, F1 score or confusion matrix, but these metrics are unsuitable for assessing a regression-based model where logarithmic loss is more suitable.

2.2.7 Data Type. This group of fixes changes the data type of inputs to match the DNN's expectation.

2.2.8 Activation. The activation function for a node in a layer of DNN maps inputs to the output. This group of fixes changes the activation function used in a layer to better match the problem.

2.2.9 Iterations. This group of fixes adjusts the number of time the training process will be run.

This is generally done to improve accuracy or to reduce overfitting. These fixes include changing batch size or epoch. In some cases, developers add a loop around the entire training process.

2.2.10 Versioning. DNN libraries are being rapidly developed, and a number of releases are not backward compatible that breaks code. This group of fixes adapts a code to work with the new version of the DNN library.

2.2.11 API Contract. When the output of a DNN API is fed to the input of another DNN API operation, these two operations have to be compatible. This group of fixes adds adapters to fix incompatibilities between composed operations.

2.2.12 Data Wrangling. Data wrangling refers to changing the form of data without changing its intent. It is generally done to fix the data for the downstream operations. This group of fixes adds data wrangling to fix a DNN, e.g. by data shifting, shuffle, etc.

2.2.13 Monitor. The fixes in this category add code for diagnostics during the training process, typically to print training statistics. This kind of fixes do not repair the flaw in the code, but they help to localize the bug.

2.2.14 Optimizer. This group of fixes modifies the optimization algorithms used by the DNN model. The optimization algorithm, which is dependent on the problem, determines the iterative process followed to improve the accuracy of the DNN model.

2.2.15 Change Neural Architecture. This group of fixes essentially re-do the DNN model because the initial model was unsuitable.

2.3 Labeling

For labeling, we used the classification scheme in Table 2. Two Ph.D. students with expertise in these DNN libraries were requested to label the fixes according to classification schemes. We held multiple training sessions to train the raters with the classification scheme. We used the Cohen's Kappa coefficient [30] to measure the agreement between the raters after the labeling of every 100 bug fix patterns. We found that the Cohen's Kappa coefficient is 82% for the first 100 labelings, 85% for the second 100 labeling. This high value of the Cohen's Kappa coefficient indicates perfect agreement between the raters. In the presence of a moderator, the repair patterns for which there is label conflict between the raters were reconciled. We adapted this methodology from [10]. Following this strategy, we labeled all the fixes and reconciled the labeling conflicts through moderated discussions. The Cohen's Kappa score throughout the process was more than 85% indicating a clear understanding and perfect agreement.

3 BUG FIX PATTERNS

In this section, we explore the answer to RQ1 to understand what are the most common bug fix patterns in DNN. To answer RQ1 we take the labeled dataset and statistical distribution of the bug fix patterns across different categories. We also analyze the source code and diffs for those fixes to understand the challenges underlying those patterns. Fig. 1 shows the distribution of different bug fix patterns in *Stack Overflow* and *GitHub*.
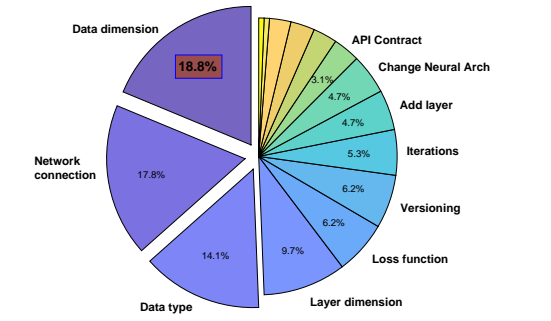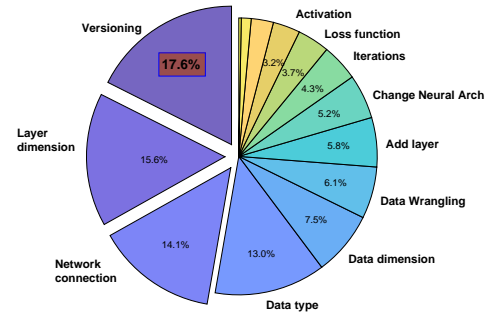
(a) Distribution of Bug Fix Patterns in *Stack Overflow* (Labels less than 3.1% are hidden)

(b) Distribution of Bug Fix Patterns in *GitHub* (Labels less than 3.2% are hidden)

Figure 1: Bug fix pattern distribution

Table 3: Bug Fixes in *Stack Overflow* (SO) and *GitHub* (GH)

	Caffe		Keras		Tensorflow		Theano		Torch	
	SO	GH	SO	GH	SO	GH	SO	GH	SO	GH
Loss function	11.1%	0.0%	6.3%	1.2%	4.2%	7.8%	13.3%	6.25%	5.9%	4.9%
Neural connection	14.8%	11.8%	18.9%	10.2%	22%	13.3%	0.0%	21.9%	0.0%	26.8%
Add layer	11.1%	11.8%	5.6%	9.6%	2.5%	1.1%	0.0%	0.0%	5.9%	2.44%
Layer dimension	3.7%	0.0%	7.0%	26.3%	13.6%	3.3%	13.3%	9.4%	11.8%	9.8%
Data dimension	22.2%	0.0%	22.4%	9.6%	11.9%	2.2%	26.7%	15.6%	23.5%	7.3%
Accuracy metric	0.0%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Data type	3.7%	29.4%	7.7%	13.8%	19.5%	10.0%	26.7%	6.2%	29.4%	14.6%
Activation	7.4%	0.0%	3.5%	3.6%	0.8%	0.0%	6.7%	12.5%	0.0%	2.4%
Iterations	7.4%	5.9%	4.95%	3.6%	5.9%	4.4%	0.0%	9.4%	5.9%	2.4%
Versioning	0.0%	0.0%	6.3%	9.0%	8.5%	51.1%	6.7%	0.0%	5.9%	0.0%
API contract	3.7%	0.0%	1.2%	1.2%	5.1%	1.1%	0.0%	3.1%	0.0%	0.0%
Data wrangling	0.0%	35.3%	4.2%	2.4%	1.7%	1.1%	0.0%	6.2%	0.0%	19.5%
Monitor	11.1%	5.9%	2.8%	1.2%	1.7%	4.4%	0.0%	0.0%	0.0%	4.9%
Optimizer	0.0%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.4%
Change neural arch.	3.7%	0.0%	5.6%	8.4%	2.5%	0.0%	6.7%	9.4%	11.8%	2.4%

3.1 Data Dimension

> **Finding 2** ⇒ Fixing data dimension is the most common bug fix pattern (18.8%) in *Stack Overflow* that can affect the robustness of DNN model.

A large number of bugs (59 out of 415) in *Stack Overflow* are fixed by changing data dimension. This suggests that most DNN models can easily be broken if the data processing pipeline changes or a different format of data is fed to the DNN. For example, in the following code snippet, we see how the bug discussed in a *Stack Overflow* post[1] is fixed by adding a dimension to the input images.

```
model = Sequential()
...
model.compile()
model.load_weights('./ddx_weights.h5')
img = cv2.imread('car.jpeg', -1) # this is is a 32x32
    RGB image
img = np.array(img)
+ img = img.reshape((1, 3, 32, 32))
y_pred = model.predict_classes(img, 1)
print(y_pred)
```

---

[1] https://stackoverflow.com/questions/37666887

In the listing the developer wants to read a CIFAR-10 image whose dimension is (32,32,3) but the expected image size was (1,3,32,32). Data dimension change can be categorized into the following kinds.

Resize. Resizing the input data is common, e.g. resizing an input image of shape (190,150) to (150, 150). A risk in this kind of fix is the loss of information from the input due to resizing. Surprisingly, this risk is never stated in the fixes presented on the bug fixes we have studied. 11 out of the 59 data dimension fixes involve resizing the data. Resizing can be done in two ways: downscale or upscale. The downscale is the method where the risk due to data loss is critical from our observation. Upsampling does not have this risk of data loss and recent results suggest that adding noise to the data can potentially increase the robustness of a model [32].

> **Finding 3** ⇒ 63% of the resize related posts in *Stack Overflow* utilize the downscaling that can decrease the robustness of a DNN.

7 out of the 11 data resizing post in *Stack Overflow* involves downscaling. Downscaling decreases the robustness and [31] has shown that a simple resize downscaling operation can have a negative impact on the robustness. During downscaling, significant information loss occurs and that eventually decrease the features learned by the DNN. A DNN trained with downscaled images will be easier to attack compare to one trained with original images. Our findings suggest that it would be useful to verify the effect of the resizing fix on vulnerability of the DNN.

Reshape. Reshaping the input occurs when the input vector shape is changed. For example, a vector of size (32, 32) is changed to (1,32,32). In this case, no data loss happens and the tensor order is changed from 2D to 3D. An example of this fix is presented in the *Stack Overflow* post #41563720[2]. The reshaping does not lead to data loss. 38 out of 59 data dimension fixes involve reshaping the dimension of the input. Reshape may also involve changing the dimension through one hot encoding like the following code snippet to fix *Stack Overflow* post #49392972[3]:

```
train_labels = to_categorical(train_labels)
```

---

[2] https://stackoverflow.com/questions/41563720
[3] https://stackoverflow.com/questions/49392972

Reorder. To make this kind of dimension change, the input data is ordered mostly to change the channel position. In image classification problems channel refers to the color channels of three primary colors. (height, width, channel) represents the typical structure of a 3D image. For example, the input of shape (32,32,3) is changed to (3,32,32) to fix some bugs. Here the channel number is moved to the first argument from the third argument. It can also involve changing the image dimension order format like from RGB to BGR as in the following snippet for fixing *Stack Overflow* post # 33828582[4]:

```
img = caffe.io.load_image( "ak.png" )
+ img = img[:,:,::-1]*255.0 # convert RGB->BGR
```

> **Finding 4** ⇒ Reorder and reshaping (79.7% of the data dimension fixes in *Stack Overflow*) need understanding of the specifications of the DNN layers as well as the libraries.

9 out of 59 data dimension fixes involve reordering the dimension of inputs. This is done because some of the libraries require dimension in a specific order. These fixes are seen in the bugs where the developer works with multiple libraries having different channel position requirements in the image data, such as *Stack Overflow* post #45645276[5]. DNN training can be assumed as a gradient descent based optimization problem, which can be computed when all the functions utilized in the model creation are differentiable. Data should be changed in such a fashion that does not affect the gradient descent computation to avoid side effects. In *reshape* and *reorder*, the only changes occur is the addition of dimension and reordering of the values that do not impact the gradient descent computation. So these changes theoretically have no side effects in the DNN models' behavior.

3.2 Layer Dimension

> **Finding 5** ⇒ In *GitHub* layer dimensions fixes are used more frequently (15.6%) to fix the crash related bugs (75.9%).

In *GitHub*, data dimension related fixes involve 7.5% of all the fixes. On the other hand, fixing the layer dimensions to make the DNN compatible with input data is a more common practice in *GitHub*. Dimension related fixes can be done by analyzing the input and output of the layers by converting a neural network into a data flow graph. This kind of fixes includes dimension reduction or addition based on the adjacent layers' structure. However, these fixes can be either done by changing the data dimension to match the data with the layer dimension or vice-versa. The choice of the fix has an impact on the performance of the model. This phenomenon is known as the *curse of dimensionality* [8]. The curse of dimensionality states that increasing or decreasing the dimension can lead to overfitting/underfitting problems. PCA [15], T-SNE [19] are some examples of the dimension reduction techniques that reduce the dimension of the features but these techniques suffer from the curse of dimensionality. To build an automated approach to avoid this side effect, a tool needs to optimize the performance of the

---

[4]https://stackoverflow.com/questions/33828582
[5]https://stackoverflow.com/questions/45645276/

model by either changing the data dimension or the layer dimension. AutoML [14] has done some preliminary work in this field that restructures the model by changing the layer dimension and adding layers to increase the performance. To the best of our knowledge, no tool currently exists that analyzes both data dimension and layer dimension changes to pick the optimum operations for a DNN model.

3.3 Version-related Fixes

> **Finding 6** ⇒ Versioning-related bug fixes are the highest (17.6%) in *GitHub* indicating the high maintenance cost in DNN software due to library versioning.

We have found that in *GitHub*, long-running projects have to fix a lot of bugs due to frequently changing versions of the DNN libraries. A number of these fixes require changing the API signatures to match with changes in the libraries. We have also observed a more complex fix pattern for projects that use *Tensorflow* library as discussed in §7.3. *Tensorflow* often makes invasive, backward-incompatible changes adding difficulties to fix the introduced bugs. This indicates that the maintenance cost in DNN software is high.

3.4 Network Connection

> **Finding 7** ⇒ Network Connection is a prevalent fix in both *Stack Overflow* (17.8%) and *GitHub* (14.1%) to fix crash (57.14%), incorrect functionality (16.19%), and bad performance (12.38%) effects.

The tensor and data flow through the network in a DNN during forward and backward propagation or prediction. For a smooth flow of data, the end to end connectivity in the network is essential. 57 out of 415 fixes require fixing or adjusting the connectivity in the network. We have found three kinds of network connection repairs.

Merge layers. A number of repair cases fixed bugs by merging two parallel layers into a single layer. For example, the following code snippet shows a fix,

```
+ main_branch.add(Merge([branch_1, branch_2], mode = 'dot'))
```

where two different branches are connected through dot product. The network was disconnected in the bug leading to a crash.

Add feedback loops and input layers. In some bug fixes, a feedback loop is added in the DNN model. In some of the fixes, the model is connected to the input via an input layer like the following:

```
+ lstm_out = LSTM(128, input_shape=(maxlen, len(chars)))(net_input)
```

Transfer learning. Transfer learning is a popular technique that takes an already-trained network with a different dataset. Then, the new model modifies the last layers to support the goal of the new problem and then performs some retraining without modifying the weights and biases of the layers from the previous network. We have observed several network connection fixes needed when the developer is attempting transfer learning. Generally these fixes

change the last few layers of the DNN. One such kind of fix is shown below from *Stack Overflow* post #57248121[6]:

```
+ model_final.fit_generator(train_generator.flow(np.
    array(X_train), np.array(y_train), batch_size=32),
+ validation_data=test_generator.flow(np.array(X_test),
    np.array(y_test), batch_size=32),
+ steps_per_epoch=len(X_train)/32,
+ validation_steps=len(X_test)/32,
+ epochs=50)
```

In this example, the developer wants to train the imagenet with a pretrained network VGG19 that has been used for face recognization. In this bug, the developer does not provide the correct data input size that leads to an error and fix was to include a data generator that loads the training data as expected by the VGG19 model.

3.5 Add Layer

> **Finding 8** ⟹ 30% of the add layers related fixes in *Stack Overflow* includes adding Dropout layer that can increase the training time ~2-3x.

In a DNN model, adding layer helps to increase the performance and learn the features more accurately. We have found that a vast majority (~30%) of these bug fix patterns includes the addition of the dropout layer. Dropout layer helps in removing the effect of the overfitting [26] that can also be achieved by using backpropagation. According to [26], backpropagation works better for the training dataset but does not work for new data. Dropout layer randomizes the structure of the architecture that helps the neural network to learn more features with every iteration. Dropout layer removes the connection between layers randomly stopping the data flow through those nodes and edges. Randomly reducing connections can have a negative impact on training time. With Dropout layers, the convergence of the training takes ~2-3x more time [26].

3.6 Loss Function

> **Finding 9** ⟹ Among DNN hyperparameters, change of loss function happens to fix 6.2% (highest) of the bugs in *Stack Overflow* and 3.7% in *GitHub* that helps to enhance prediction accuracy and increase the robustness against adversarial attacks.

Loss function plays a crucial role in the convergence of the training process and in getting better accuracy during prediction. A model with wrong loss function does not learn the decision boundary of the features well and there can be overlap between the decision boundaries [12, 21] in the high dimensional feature space making the model vulnerable to adversarial attacks [24]. By a careful and deeper analysis of these loss function related fixes, we have found that they can be categorized into the following kinds:

Add new loss function: The fixes in this category involve adding a custom or built-in loss function. 10 out of 23 fixes fall into this category. In some of the fixes, it is needed to modify the network connectivity for the new loss function to work. For example, in the following fix of the bug in *Stack Overflow* post #51257037[7], the

last layer is kept outside the gradient descent computation during training by adding trainable = False.

```
output = Dense(1, trainable = False)(hidden_a)
```

The custom loss function was designed by the developer in such a way that all but the output layer participate to lead to the convergence of the model. However, the convergence was not successful, as the output layer was actively participating in the forward and backward propagation that caused an abrupt change in the value of the loss function. **Fixing these bugs require live trainable parameter analysis, which is similar to live variable analysis in SE. This approach will help to monitor the active trainable parameters during the training to localize and fix these bugs.** Currently, the developer needs to rely on theoretical knowledge to fix these bugs due to the lack of such kind of analysis frameworks. **Change loss function:** 9 instances of bug fixes fall into the category of changing the loss function. Our analysis of these fixes reveals that the choice of these loss functions is sometimes confusing. Developers need to understand the data properties and the goal of the DNN task to come up with a proper loss function. For example, in constructing DNN models for classification problems, the developers are confused between the choice of binary_crossentropy and categorical_crossentropy as discussed in the fix of *Stack Overflow* post #45799474[8] and *Stack Overflow* post #42081257[9]. The first loss function works better for the binary classification problems; however, when the classification problem has more than two categories, one should use categorical_crossentropy as a loss function to avoid poor performance. Sometimes, the fix involves adding some filter to the mathematical operation used in the loss function. For example, we see the following bug fix of *Stack Overflow* post #34223315[10]

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.
    clip_by_value(y_conv, 1e-10,1.0)))
```

caused by the following line:

```
cross_entropy = -tf.reduce_sum(y_ * tf.log(y_conv))
```

In the above code snippet, the problem is that the user will get NaN values if y_conv becomes negative as the log of negative numbers is undefined. The fix adds a clipper function to filter out negative values to the log operation. In another fix of the same kind of bug in *Stack Overflow* post #42521400[11], softmax is used as the filtration operation that stops propagating values <= 0 to the log operation:

```
softmax = tf.nn.softmax(logits)
xent = -tf.reduce_sum(labels * tf.log(softmax), 1)
```

3.7 Commonality of Fix Patterns in *Stack Overflow* and *GitHub*

> **Finding 10** ⟹ The p-value is 0.83 between the bug fix pattern distributions of *Stack Overflow* and *GitHub* indicating commonality of bug fix patterns in *Stack Overflow* and *GitHub*.

[6]https://stackoverflow.com/questions/57248121/
[7]https://stackoverflow.com/questions/51257037/

[8]https://stackoverflow.com/questions/45799474/
[9]https://stackoverflow.com/questions/42081257
[10]https://stackoverflow.com/questions/34223315/
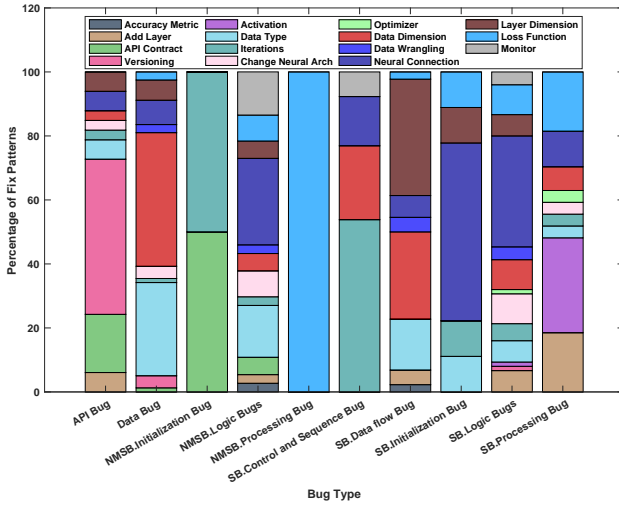[11]https://stackoverflow.com/questions/42521400/

Figure 2: Distribution of Bug Fix Patterns for Different Bug Types *Stack Overflow*
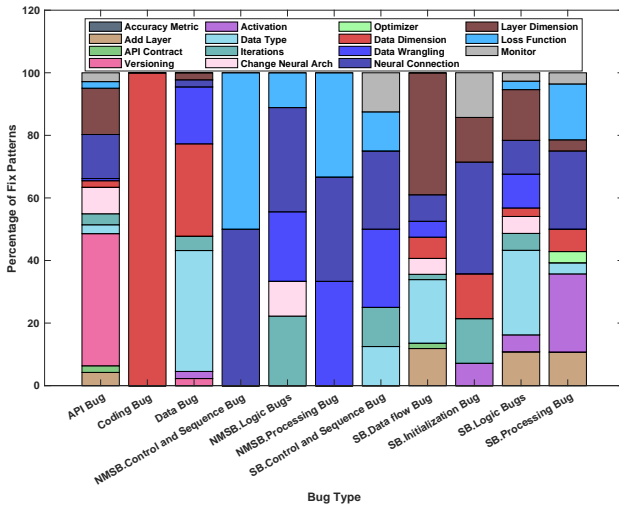

Figure 3: Distribution of Bug Fix Patterns for Different Bug Types *GitHub*

We have conducted a t-test at 95% significance level to understand the distribution of bug fix patterns in *Stack Overflow* and *GitHub*. The null hypothesis is H_0 : *the distributions are the same*. The null hypothesis is to be rejected if the p-value is less than 5% or 0.05. Our computation shows that the p-value is very high (0.83). So, H_0 can not be rejected concluding that the distributions are similar. We also notice that though in some bug fix categories e.g., data dimension, layer dimension, and versioning, there is a significant difference among the *Stack Overflow* and *GitHub* distributions, the other categories have a similar share of occurrences in *Stack Overflow* and *GitHub*. This indicates that the bug fix patterns have commonality across *Stack Overflow* and *GitHub*.

Figure 4: Fix of *Stack Overflow* #49742061

4 FIX PATTERNS ACROSS BUG TYPES

To answer RQ2, we analyze the correlation between the bug types in the bug dataset presented by [10] and the bug fix patterns studied by this paper using the distribution of the bugs and their corresponding fixes. The distribution of bug fix patterns across different bug types in *Stack Overflow* and *GitHub* are shown in the Fig. 2 and 3, respectively. The horizontal and the vertical axes describe the different bug types from [10] and the percentage of different fix patterns needed to fix those bugs, respectively.

> 💡 **Finding 11** $\Rightarrow$ For API bugs, fixing of the specifications between APIs is dominant (42% in *Stack Overflow* and 48% in *GitHub*) .

Fixing API specifications involves changing API contracts due to API versioning and supporting inter-library operations within a model. Fixing API specifications is needed due to the following reasons:

Change of specifications due to version upgrade: 20 fixes in *Stack Overflow* involve changing specifications which are required due to the change of the library version. The changes during the upgrade of the library version involves the following changes: **change fully qualified method names**, **change API signature** and **change probabilistic behavior of the APIs**. Though fixes due to the change of fully qualified method names and change of API signature are well-studied problems [3, 6, 13], the fixes due to the change of probabilistic behavior of the APIs are hard and different from traditional API changes. Localizing of these bugs are difficult due to the lack of sophisticated probabilistic analysis tools for DNN. For example, the bug discussed in *Stack Overflow* #49742061[12] says that the results are different in two versions of *Tensorflow*. The fix of this bug involves adding a dead code line that tweaks around the underlying probabilistic behavior of the APIs by overriding the modified random seed. The fix of *Stack Overflow* #49742061 is shown in Fig. 4. The fix adds the line `Xf = tf.contrib.layers.flatten( X )` before the line `R = tf.random_uniform( shape = () )`. This addition overrides the random seed in the new version with the one in the previous version.

Our observation gives the intuition that the fix of versioning bugs due to the change of the probabilistic distribution in different version needs new DNN specific probabilistic analysis techniques.
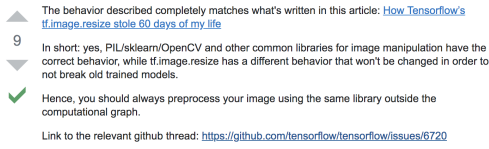
---

[12]https://stackoverflow.com/questions/49742061/

The behavior described completely matches what's written in this article: How Tensorflow's tf.image.resize stole 60 days of my life

9

In short: yes, PIL/sklearn/OpenCV and other common libraries for image manipulation have the correct behavior, while tf.image.resize has a different behavior that won't be changed in order to not break old trained models.

✓ Hence, you should always preprocess your image using the same library outside the computational graph.

Link to the relevant github thread: https://github.com/tensorflow/tensorflow/issues/6720

Figure 5: Fix of *Stack Overflow* #54497130

Change specification to support interlibrary: In these fixes, the DNN program uses more than one library. These bugs arise due to the similar assumption of the behavior and specifications for different APIs in different libraries. Fixing of these bugs requires the expertise in both the libraries e.g., the bug discussed in *Stack Overflow* #54497130[13] that is shown in Fig. 5. The discussion points to an issue in the official *Tensorflow* repository. The solution suggested to avoid using APIs from other libraries to pre-process images. However, in similar scenarios, the use of specialized image processing libraries is recommended to get better performance.

From Fig. 2 and 3, we have found that fixing the data dimension is the most prominent pattern (41.77%) for fixing data bugs in *Stack Overflow*. For fixing data bugs in *GitHub*, the most prominent fix patterns are the change of data type (38.64%) and data dimension (29.55%). This suggests that for fixing data bugs, the major changes are related to data dimensions. This happens because the dimension of the data is very important for the correct functionality of the DNN model.

For fixing logic bugs the most common practice is to modify the network connectivity (~27.03% in *Stack Overflow* and ~33.33% in *GitHub*). A detailed discussion on network connectivity is presented in §3.4. Whereas, a significant amount of data flow bugs can be fixed by changing the layer dimension (~36.36% in *Stack Overflow* and ~38.98% in *GitHub*). A detailed discussion on fixing layer dimension is presented in §3.2.

These observations give us the intuition that for fixing different types of bugs, unique technical approaches might be needed.

5 FIX PATTERNS ACROSS LIBRARIES

To answer RQ3, we have studied the distribution of fix patterns across the 5 libraries. Then, we have conducted statistical pairwise t-test at 95% significance level between the libraries. Table 4 shows the p-values found from this test across the libraries.

Table 4: P-value of the distribution of Bugs btween the libraries

Library	Caffe	Keras	Tensorflow	Theano	Torch
Caffe	1.0	0.0045	0.00735	0.19	0.30
Keras	0.0045	1.0	0.84	0.0021	0.0024
Tensorflow	0.0073	0.84	1.0	0.0039	0.0044
Theano	0.19	0.0021	0.0039	1.0	0.80
Torch	0.30	0.0024	0.0044	0.80	1.0

We assume the null hypothesis is H_0: *the distribution of the fix patterns across two libraries are same.* If the p-value is less than 5% or 0.05, then we reject H_0. The p-value for the library pairs *Caffe-Theano* (.19), *Caffe- Torch* (.30), *Keras- Tensorflow* (0.84), *Theano-Torch* (0.8) are much greater than 5%. So in these, cases we can

[13]https://stackoverflow.com/questions/54497130/

not reject the null hypothesis. So, the libraries *Caffe*, *Theano* and *Torch* show similar kind of bug fix patterns. And *Keras- Tensorflow* form a very strong related group with a p-value close to 100%. This suggests that similar kind of automatic bug fix tools may be reused for *Caffe*, *Theano*, and *Torch* after converting into a common intermediate representation. Similarly, *Keras* and *Tensorflow* bug fixes can be entertained by a similar technical approach.

6 INTRODUCTION OF BUGS THROUGH FIXES

> 💡 **Finding 12** ⟹ 29% of the bug fixes introduce new bugs in the code adding technical debt [25] and maintenance costs.

To answer RQ4, we have analyzed 100 randomly chosen fixes from *Stack Overflow* to understand whether fixing a bug can introduce a new bug. We have read the replies to the answers selected by filtering criteria discussed in §2. Then, we have identified whether the fix introduced new bugs by analyzing all replies to the answer fixing the original bug and classify them into bug type, root cause, and impact using the classification scheme proposed by the prior work [10]. We have found that 29% fixes in the randomly sampled dataset introduce at least one new bug in the code. Furthermore, we have compared the bug type, root cause, and the effect of the bugs of *Stack Overflow* posts with the newly introduced bugs and have found that 6.8%, 13.8%, and 24.1% of the bugs match the classification of the parent bug type, root cause, and impact, respectively. This result depicts that a majority of the bugs introduced are of new types and their behavior is entirely different than that of the parent bugs'. In the Table 5, we have shown the distribution of the new bugs across the different libraries and how these new bugs are classified into different categories of bug type, root cause, and impact. We have also found that the *Crash* (55.8%) is the most common impact of these new bugs and a majority of these bugs are of *API Bug* (37.9%), and the most common root cause of these bugs are *API Change* (34.5%). 44.8% and 34.5% of the newly introduced bugs are from *Keras* and *Tensorflow*. *Caffe*, *Theano*, and *Torch* related bug fixes introduce 10.34%, 3.45%, and 6.90% new bugs respectively.

> 💡 **Finding 13** ⟹ 37.9% of the new bugs are from API Bug, 34.5% of them are due to API Change, and 55.2% of them end in a crash.

In Fig. 6, the relation between the parent bugs' root cause, type and effect with the newly introduced bugs' distribution have been visualized. In this visualization, the *old* represents the parent bug and the relation has been drawn by a connection between two bug distributions. The width of the connection determines the strength of the relation. The perimeter covered by each bug type/root cause/impact depicts its overall strength. We have found that a large section of bug fixes introduces API bug and the major reason for that is the API change that mostly due to the versioning of the APIs and these fixes primarily lead to a crash and bad performance.

7 CHALLENGES IN FIXING BUGS

In this section, we explore the answer to RQ5 to identify the common challenges faced by the developers in fixing the DNN bugs. To
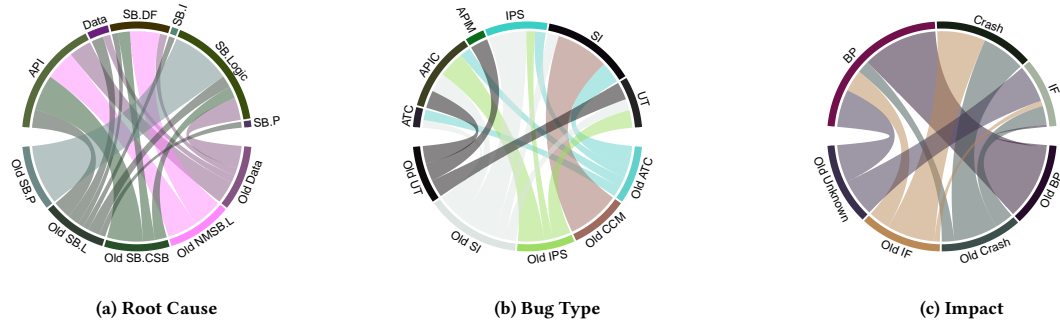
(a) Root Cause (b) Bug Type (c) Impact

Figure 6: Bug fix pattern distribution: SB.P→SB.Processing, SB.L→SB.Logic, DF→Data Flow,SB.I→SB.Initialization, ATC→Absence of Type Checking, BP→Bad Performance, IF→Incorrect Functionality

Table 5: Statistics of the Introduction of New Bugs During Bug Fix

Library	Bug Type						Root Cause						Impact		
	API Bug	Data Bug	SB.DF	SB.I	SB.L	SB.P	ATC	APIC	APIM	IPS	SI	UT	Bad performance	Crash	IF
Caffe	0%	0%	0%	0%	100.0%	0%	0%	0%	0%	33.3%	66.7%	0%	66.7%	0%	33.3%
Keras	30.8%	7.69%	30.7%	0%	23.1%	7.69%	7.69%	30.8%	0%	15.4%	15.4%	30.8%	23.1%	61.5%	15.4%
Tensorflow	60.0%	20.0%	0%	10%	10.0%	0%	20%	60%	0%	10%	10%	0%	20%	70%	10%
Theano	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	100%	0%	100%	0%	0%
Torch	50.0%	0%	50%	0%	0%	0%	0%	0%	50%	0%	0%	50%	0%	50%	50%

understand the challenges, we have used a classification scheme separate from bug fix patterns. Similar to the labeling performed for bug fix patterns, two raters have independently classified each post used in this study. These classes of new challenges are described below:

7.1 DNN Reuse

Training DNN models can be expensive because it requires sophisticated computational resources and a large amount of labeled data which might not be readily available. This has led to the reuse of DNN models that creates unique issues such as backdoor attack [4], injection of bias [2], and mismatch between the intent of the pretrained DNN model and the intent of the developer.

```
base_model = ResNet50(input_shape=(224, 224, 3), include_top=False, weights='
    imagenet', pooling='avg')
+ x=base_model.output
+ x = Dense(512, activation='relu')(x) #add new layer
+ x = Dropout(0.5)(x) #add new layer
+ x = Dense(512, activation='relu')(x) #add new layer
+ x = Dropout(0.5)(x)
```

In the example above from *Stack Overflow* post # 49226447[14], the developer wants to train a predefined DNN model structure `ResNet50` using the cancer dataset. The trained network results in overfitting as the developer was not aware of the structure of the reused model and needed to modify the network by adding dropout and dense layers to reduce the effect of overfitting.

7.2 Untraceable or Semi-Traceable Error

In case of a crash bug, the common strategy to localize the bug is to analyze the error message. However, we have found that bug localization is very challenging in DNN software because errors

and faults are non-trivially related. To illustrate, consider the code snippet below from *Stack Overflow* post # 33474424[15]:

```
model = Sequential()
model.add(Dense(hidden_size, input_dim=input_size, init='uniform'))
model.add(Activation('tanh'))
...
y_pred = model.predict(X_nn)
```

This code produces the following error trace:

```
ttributeError                    Traceback (most recent call last)
<ipython-input-17-e6d32bc0d547> in <module>()
1
----> 2 y_pred = model.predict(X_nn)
491      def predict(self, X, batch_size=128, verbose=0):
492          X = standardize_X(X)
--> 493          return self._predict_loop(self._predict, X, batch_size,
    verbose)[0]
494
495      def predict_proba(self, X, batch_size=128, verbose=1):

AttributeError: 'Sequential' object has no attribute '_predict'
```

From this error message, a developer might start digging into the code of `predict` function and the `Sequential` object; however, the issue is the missing compilation step. Due to this, the model connection is not initialized and error propagates to the `predict` operation and halts the training process. We have studied randomly 50 bugs yielding `Crash` from *Stack Overflow*. **We have found that 11 out of 50 posts does not indicate any error message and in rest of the 39, 20 posts have a fix that does not match with the error message.**

7.3 Fast and Furious Releases

We have previously discussed that a large number of fixes are due to the rapid versioning and invasive changes in DNN libraries.

To study this challenge, we have labeled all removed, reconfigured, or renamed operations of *Tensorflow* from version 1.10 to 2.0 (latest in June 2019). In Table 6, we have shown the number of

---

[14]https://stackoverflow.com/questions/49226447

[15]https://stackoverflow.com/questions/33474424

Table 6: *Tensorflow* API changes. Change= # of operations changed in comparison to the previous version.

Version	# of Symbols	Change	Release Date [28]
v2.0 (Beta)	6504	2185	Jun 7, 2019
v1.14.0	8363	59	Jun 18, 2019
v1.13.1	3560	39	Feb 25, 2019
v1.12.0	3314	52	Nov 1, 2018
v1.11.0	3145	175	Sep 25, 2018
v1.10.0	3230	N/A	Aug 7, 2018

symbols of operations available for each *Tensorflow* releases and the number of operations that have been deleted, renamed, or re-configured in comparison to the previous version. **We have found that from the v1.14 to v2.0 26% of the operations have been changed**.

A non-trivial challenge for repairing DNN software is the proba-bilistic behavior of the APIs. Some of these version upgrades also change the probabilistic behavior of the APIs causing some diffi-cult bugs. An example is presented below where the change of the probabilistic distribution changes the output of the same operation with different versions[16].

```
with Tensorflow 1.3
Z3 = [[−0.44670227 ...   0.46852064]
[−0.17601591 ...   0.5747785 ]]
with Tensorflow 1.4+
Z3 = [[ 1.44169843 ...   1.36546707]
[ 1.40708458 ...   1.26248586]]
```

8 THREATS TO VALIDITY

External Threat A source of external threat can be the dataset used to study the bug repair pattern. To alleviate this threat we use a benchmark dataset of DNN bugs prepared by [10].

Internal Threat An internal threat can be the coding approach used to classify the bug fix patterns. We use the widely adopted open coding approach to come with a classification scheme to minimize the threat. Two Ph.D. students independently came up with the classification schemes. Then, these schemes were merged through moderated discussions. The expertise of the raters can be another source of an internal threat. We alleviate this threat by involving raters who have expertise in both the DNN libraries and the bug fix patterns. The raters were also trained on the coding scheme before the labeling. We also use Cohen's kappa coefficient to measure the inter-rater agreement throughout the labeling process. And the value of kappa coefficient indicates that the labeling was successful with a perfect agreement.

9 RELATED WORKS

The closest related works are that by Zhang *et al.* [35], Islam *et al.* [10] and Pan, Kim, and Whitehead [20].

Zhang *et al.* [35] have studied bug patterns in *Tensorflow* using both *GitHub* and *Stack Overflow*. They have discussed the new patterns and characteristics of the bugs by *Tensorflow* users to write DNN applications. They have also discussed the three new challenges in detecting and localizing these bugs. The study was limited to *Tensorflow* and also does not discuss the bug fix patterns. We generalize to a number of deep learning libraries and identify the new patterns of fixing the bugs in DNN software. We also discuss

the new challenges in fixing these bugs. We have discussed three new challenges in fixing DNN bugs.

Islam *et al.* [10] have studied five different DNN libraries. They have done a more general study on DNN bug characteristics be-yond *Tensorflow*. However, their work has not discussed the bug fix patterns and challenges in fixing those bugs. Our work focuses on the identification of the bug fix patterns and challenges in those fixes.

Pan, Kim, and Whitehead [20] have studied seven Java projects to discuss the bug fix patterns in these projects. They have also proposed a classification scheme in categorizing the bug fix patterns in Java. The classification includes 9 broad categories and a total of 26 lower-level categories. This prior research suggests that the IF-related and Method Call (MC) related bugs are most frequent. In DNN bug fix strategies, the MC and sequence addition or deletion related bug fix pattern is present. We do not find any evidence of other bug fix strategies in DNN and that has inspired us to derive a classification scheme using the open coding scheme to classify the DNN bug fix patterns.

Sun *et al.* [27] studied the issues in 3 ML libraries *scikit-learn*, *Caffe*, and *paddle* to understand the bug fix patterns in these li-braries. However, we study the DNN models created using DNN libraries.

Zhang *et al.* [34] studied 715 *Stack Overflow* bug related posts for TensorFlow, PyTorch, and Deeplearning4j to classify the ques-tions into 7 different categories and built an automated tool that categorizes questions based on the frequently found words from each category and computing the tf-idf value with respect to the keywords. Also, the authors have studied the challenges of answer-ing the question in *Stack Overflow* by calculating the response time for each category and have found 5 categories of root causes for the bug related posts. Whereas, our study has been on the bug fixing strategies for 5 DNN libraries.

Programming bugs are a well-studied field in software engineer-ing. There is a rich body of empirical studies on bugs, e.g. [5, 7, 16–18, 23, 33] and bug repair, e.g. [1, 20, 22, 36]; however, these works have not studied DNN bugs and repairs that have their own set of unique challenges [10, 11, 29, 35].

10 CONCLUSION AND FUTURE WORK

The widespread adoption of deep neural networks in software sys-tems has fueled the need for software engineering practices specific to DNNs. Previous work has shown that like traditional software, DNN software is prone to bugs albeit with very different charac-teristics. It is important to further understand the characteristics of bug fixes to inform strategies for repairing DNN software that has these bugs. How do developers go about fixing these bugs? What challenges should automated repair tools address? To that end, we conducted a comprehensive study to understand how bugs are fixed in the DNN software. Our study has led to several findings. First of all, we find that bug fix patterns in DNN are significantly different from traditional bug fix patterns. Second, our results show that fixing the incompatibility between the data and the DNN alone can be of significant help to developers of DNN software, especially if the developers can be warned about the impact of their bug fix on the robustness of the DNN model. Third, our study shows that a

---

[16]https://stackoverflow.com/questions/49742061

prevalent bug fix pattern is version upgrade. While version upgrade is well-studied in SE research, our bug fix patterns suggest that automated repair tools will need to address at least two unique challenges: invasive, backward incompatible changes and probabilistic behavior change. Fourth, our study shows that the structure of the DNN itself needs to be represented in repair tools because several fix patterns rely on identifying incompatibilities in that structure. For instance, network connection fixes where disconnected layers are identified and connected, or adding missing layers, etc. Fifth, we have found that a significant number of bug fixes introduce new bugs in the code. Finally, we have identified three challenges for fixing bugs: bug localization is very difficult, reuse of the DNN model is hard because of limited insights into its behavior, and keeping up with rapid releases is hard.

This study opens up several avenues for future work. First and perhaps most immediately, a number of bug fix patterns identified by this work can be automated in repair tools. Such tools for bug repairs can help the developers integrating DNN into their software. Second, an abstract representation of the DNN along with the code that uses it can be developed. We saw several bug fix patterns that rely on analyzing such a representation. Third, there is a critical need to improve bug localization for DNN by addressing unique challenges that arise, and by creating DNN-aware bug localization tools. Fourth, and somewhat surprisingly to us, there is an urgent need to detect bugs introduced by dimension mismatch and specially changes that have the potential to introduce vulnerabilities in the DNNs. Fifth, urgent work is needed on upgrade tools that encode the semantics of version changes and keep up with the change in the signature and semantics of DNN libraries. This is important to keep pace with rapid development in this area.

REFERENCES

[1] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 97–106.

[2] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[3] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 255–265.

[4] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 73–88.

[6] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 64–73.

[7] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82–101.

[8] Jerome H Friedman. 1997. On bias, variance, 0/1âĂŤloss, and the curse-of-dimensionality. *Data mining and knowledge discovery* 1, 1 (1997), 55–77.

[9] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[10] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. 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 (ESEC/FSE 2019)*. ACM,

[11] Md Johirul Islam, Hoan Anh Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. What Do Developers Ask About ML Libraries? A Large-scale Study Using Stack Overflow. *arXiv preprint arXiv:1906.11940* (2019).

[12] Katarzyna Janocha and Wojciech Marian Czarnecki. 2017. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659* (2017).

[13] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How APIs break–an empirical study. *Information and Software Technology* 65 (2015), 129–146.

[14] Haifeng Jin, Qingquan Song, and Xia Hu. 2018. Auto-keras: Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282* (2018).

[15] Ian Jolliffe. 2011. *Principal component analysis*. Springer.

[16] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 25–33.

[17] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 329–339.

[18] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 381–392.

[19] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.

[20] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.

[21] Rangeet Pan, Md Johirul Islam, Shibbir Ahmed, and Hridesh Rajan. 2019. Identifying Classes Susceptible to Adversarial Attacks. *arXiv preprint arXiv:1905.13284* (2019).

[22] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 40–49.

[23] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs. jar: A large-scale, diverse dataset of real-world Java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 10–13.

[24] Sean Saito and Sujoy Roy. 2018. Effects of Loss Functions And Target Representations on Adversarial Robustness. *arXiv preprint arXiv:1812.00181* (2018).

[25] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. (2014).

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[27] Xiaobing Sun, Tianchi Zhou, Gengjie Li, Jiajun Hu, Hui Yang, and Bin Li. 2017. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 348–357.

[28] TensorFlow. 2019. TensorFlow Github. https://github.com/tensorflow/tensorflow/tags/.

[29] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.

[30] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[31] Qixue Xiao, Kang Li, Deyue Zhang, and Yier Jin. 2017. Wolf in Sheep's Clothing-The Downscaling Attack Against Deep Learning Applications. *arXiv preprint arXiv:1712.07805* (2017).

[32] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. 2017. Mitigating adversarial effects through randomization. *arXiv preprint arXiv:1711.01991* (2017).

[33] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2011. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*. ACM, 93–102.

[34] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE.

[35] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 129–140.

[36] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 913–923.

New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955