MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection

Shiqing Ma
Purdue University, USA
ma229@purdue.edu

Yingqi Liu
Purdue University, USA
liu1751@purdue.edu

Wen-Chuan Lee
Purdue University, USA
lee1938@purdue.edu

Xiangyu Zhang
Purdue University, USA
xyzhang@cs.purdue.edu

Ananth Grama
Purdue University, USA
ayg@purdue.edu

ABSTRACT

Artificial intelligence models are becoming an integral part of modern computing systems. Just like software inevitably has bugs, models have bugs too, leading to poor classification/prediction accuracy. Unlike software bugs, model bugs cannot be easily fixed by directly modifying models. Existing solutions work by providing additional training inputs. However, they have limited effectiveness due to the lack of understanding of model misbehaviors and hence the incapability of selecting proper inputs. Inspired by software debugging, we propose a novel model debugging technique that works by first conducting model state differential analysis to identify the internal features of the model that are responsible for model bugs and then performing training input selection that is similar to program input selection in regression testing. Our evaluation results on 29 different models for 6 different applications show that our technique can fix model bugs effectively and efficiently without introducing new bugs. For simple applications (e.g., digit recognition), MODE improves the test accuracy from 75% to 93% on average whereas the state-of-the-art can only improve to 85% with 11 times more training time. For complex applications and models (e.g., object recognition), MODE is able to improve the accuracy from 75% to over 91% in minutes to a few hours, whereas state-of-the-art fails to fix the bug or even degrades the test accuracy.

CCS CONCEPTS

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

KEYWORDS

Deep Neural Network, Debugging, Differential Analysis

ACM Reference Format:
Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236082

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236082

1 INTRODUCTION

Artificial intelligence (AI) especially machine learning (ML) are becoming an essential part of our daily life, evidenced by self-driving cars, Apple Face ID, digital assistants, and even the web page advertisements. Increasingly, computing systems are becoming a cohesive integration of AI models and deterministic logic just like a human is composed of the brain (for intelligence) and the body (for actions), rendering a view of *Intelligent-System* or *Intelligent-Software* that is substantially more capable than traditional computing systems. For example, autonomous robotic systems use AI navigation together with deterministic lower level control algorithms (e.g., PID controller). AlaphaGo is a computer program that combines traditional program primitives (i.e., tree based search algorithms) and deep neural networks (NN) [102]. Deterministic push-down automata are seamlessly combined with NN in machine translation in order to handle recursive natural language structures [49, 56]. AIs are an integral part of computer games which are largely deterministic computing systems [37, 99]. Similar to traditional software component sharing and reusing, data engineers also train models and share them on model repositories [2, 29] for reuse in various applications. For example, the Python *face_recognition* package [13] has been used in a number of applications. *We envision that data engineering (e.g., AI model training, tuning, maintenance) will become an essential step of software engineering.*

Just like software inevitably contains bugs and software debugging is a key step in software development process, AI/ML models may have undesirable behaviors, which we call *model bugs* in this paper, and model debugging will be an essential step in intelligent software engineering. For example, a state-of-the-art object classification model can get only 80% accuracy on the ImageNet classification challenge [97, 105], and a state-of-art natural language processing (NLP) model can get only 73% accuracy on the Children's Book Test challenge [63, 64]. Although these models are dealing with problems with inherent uncertainty such that 100% accuracy is often not achievable, there have been research showing that much higher accuracy can be achieved for the aforementioned challenges [54, 65]. In other words, the original models have bugs. Just like that software bugs might have severe consequences, model bugs could be catastrophic. Incorrect decisions could cause financial loss (e.g., model driven stock exchange), various security and safety

issues, and even endanger human lives. For example, the newly released iPhone X uses a NN based face recognition system, known as the Face ID, to unlock the phone and authenticate purchases etc. However, it can be subverted by 3D printed faces [12]. There are many accidents reported involving self-driving cars which use neural network models to replace human drivers [26].

Model bugs can be divided into two categories. One type is caused by the sub-optimal model structures such as the number of hidden layers in a NN model, the number of neurons in each layer and neuron connectivity. We call them the *structural bugs*. A lot of existing research falls into addressing these bugs [62, 79]. The other type is caused by the mis-conducted training process (e.g., using biased training inputs). We call them the *training bugs*. Our paper focuses on training bugs. In the rest of the paper, we will simply call them model bugs. Machine learning algorithms assume that the training dataset and the real world data follow the same or similar distribution so that the trained models can use features extracted from training data to properly predict unknown real world cases. Unfortunately this is usually not the case. The distribution of real world data is in general impossible to obtain. As such, a lot of extracted features are problematic (e.g., too specific for certain inputs or do not possess sufficient discriminating capabilities). However, *model interpretability* is a well known problem in data engineering which states that extracted features are so abstract that humans cannot understand them [85, 87, 94]. As such, unlike software bug fixing, which can be achieved by directly changing source code, model bug fixing cannot be achieved by directly modifying individual weight parameter values in the model. The only way to fix model bugs is to provide more training samples trying to make the training data less biased. Existing works use generative machine learning models (such as *generative adversarial networks* (GAN)) which can generate input samples similar to real world samples [42, 57, 89, 90]. However, this has limited effectiveness (see §4) due to the many inherent challenges such as having a good mechanism to ensure the generated samples are not biased.

Software debugging has been intensively studied and there have been many highly effective methods such as delta debugging [111], fault localization [70], slicing [112], and automated bug repairs [81]. All these techniques work by first identifying the root cause of the bug through some kind of execution state analysis such as contrasting states of failing execution with those of a very similar but passing execution. As such, patching becomes very targeted and hence highly effective. However, *such a root cause identification step is missing in existing model debugging techniques*. Inspired by the success of software debugging, we propose a novel model debugging technique for neural network models. Neural network models can be considered as layers of internal state variables (called *neurons*) connected by matrices. The knowledge acquired in training is encoded in the parameter values of the matrices. During application, given an input, a sequence of matrix multiplication operations together with some specially designed thresholding functions (e.g., sigmoid [88] and softmax [46]) are applied to classify the input to one of the output labels. The layers in between the input and output layer are called the *hidden* layer, whose neurons are considered representing abstract features (e.g., shape of a nose).

Given a model bug such as poor classification accuracy for a particular output label, our technique performs state differential analysis that identifies the neurons that are responsible for the misclassification, called the *faulty neurons*. Then an input selection algorithm similar to input selection in regression testing is used to select new input samples that can have substantial influence on the faulty neurons for retraining. In particular, our technique first identifies a hidden layer that is critical for fixing the model bug, by measuring if the abstraction denoted by the layer can have substantial impact on classification accuracy. Once the layer is identified, our technique further measures the importance of individual neurons/features in the layer for correct and faulty classification results of the buggy output label. For example, assume a digit recognition model has a bug in which other digits such as 6 are often misclassified as 5. Our technique determines the neurons/features that are critical for correctly recognizing 5 and those critical for misclassifying others to 5. A differential analysis (e.g., the latter subtracts the former) is then conducted to identify the features that are uniquely important for misclassification. We then select inputs that have strong presence of such features for bug fixing. Various differential analysis and input selection strategies are developed for different kinds of bugs. Details are disclosed later in the paper.

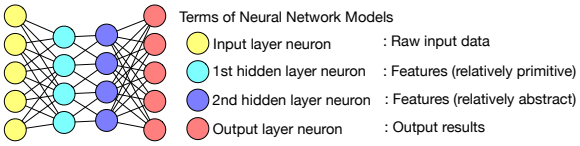


Figure 1: A Simple Neural Network Model

Our paper makes the following contributions.

- We identify that existing model debugging process is not effective due to the lack of understanding or analyzing the root causes of the model bugs.
- Inspired by state differential analysis and input selection in software debugging and regression testing, we propose a novel technique to measure importance of features for classification results, locate faulty neurons and select high quality training samples for fixing model bugs.
- We develop a prototype MODE [10] based on proposed idea, and evaluate it on 29 different models. The results show that MODE can effectively and efficiently fix all the model bugs, improving accuracy from 75% to 93% on average without introducing new bugs, whereas existing methods only work for some cases, potentially degrade the model performance, and require 11x more training time for the cases it works.

2 BACKGROUND AND MOTIVATION

2.1 Neural Network Models and Model Bugs

NN Models. NN models usually has a connected and layered structure with each layer containing a set of *neurons*. Neurons in two consecutive layers are connected, and can transmit signals through their connections. Such connections are represented by a set of matrices. The values of the matrices are called *weight parameters*. Training a neural network model is essentially to update weight parameters so that the last layer neurons can produce the expected prediction outputs. The core technique used for training is known as the *backward propagation* [96]. It calculates the gradients of output error with respect to weight parameters. Intuitively, a large gradient value indicates changing the corresponding weight can

lead to substantial output variation. The gradients are fed backward to update the matrices. Training NNs is a slow and expensive process. Hence, the training dataset is usually divided into multiple batches and the whole training consists of many epochs.

Figure 1 shows a very simple neural network model with 4 layers (left to right): *input layer, two hidden layers*, and the *output layer*. The input layer takes the raw inputs and passes them to the next layer. Hidden layers are used to extract the features of the input, such as nose shape for face recognition. Intuitively, each neuron can be considered denoting one special feature. A hidden layer constructs more abstract features from the features denoted by its preceding hidden (or input) layer. In the typical case of a fully connected NN, each neuron/feature is dependent on (connected to) all the neurons/features in the preceding layer. In general, it is very difficult for data engineers to interpret the meanings of features represented by individual neurons. After feature extraction, the output layer is trained to use the features for application specific tasks such as face recognition or object recognition.

Model Bugs. Intuitively, machine learning extracts features from a given input and compares them with existing knowledge encoded in the model to make decisions. The quality of the model highly depends on the quantity and quality of the training data. Unfortunately, it is very challenging (if not impossible) to have training data that precisely represent the distribution of real world data. As such, models usually have undesirable behaviors, which we call *model bugs* in this paper. More formally, *given a model with specific structural configuration, we say that the model is defective/buggy if its test accuracy for a specific output label is lower than the ideal accuracy, which is achieved when training inputs perfectly model the real world distribution.* Note that in this paper, we assume the model structure is given and unchangeable. We do not deal with sub-optimal model structure [71]. Instead, we investigate if better accuracy can be achieved for the given model structure. Note that many machine learning problems (e.g., face recognition) have inherent uncertainty such that 100% accuracy is infeasible. For example, even humans mis-recognize faces occasionally. Second, since the ideal accuracy is not computable, we consider *any improvement on the test score a meaningful bug fix.* The effectiveness of a bug fix is measured in two aspects: *the test score improvement* and *the efforts needed to achieve the improvement (e.g., additional training time).*

2.2 Model Debugging

Existing Model Debugging. As discussed before, the only way to fix NN model bugs is to provide more training data. To overcome the problem of lacking representative data, researchers proposed many data augmentation [1, 45] or generation techniques [57, 75], which generates new training samples that are similar to the provided input data samples. State-of-the-art generative model, *generative adversarial networks* (GANs) mainly consists of two parts: the *generator* and the *discriminator*. The generator is responsible for generating artifacts. It takes a random input (e.g., a noise vector) and changes it to something that looks real (e.g., a hand-written digit image). The discriminator is used to determine if the generated artifacts are *similar* to provided real samples. It takes the generated samples and real world samples as input, and tries to determine which ones are the real world samples. These networks are trained together. When the discriminator cannot tell the difference between real world samples and generated samples, the training terminates and the generated samples are used for training the buggy model.

Figure 2: Samples Generated by Different GANs

Unfortunately, existing model debugging techniques have limited effectiveness. They sometimes even lead to degenerated models. Firstly, generative machine learning is highly challenging and current solutions have various limitations [33, 34, 57]. For example, as shown in [33, 34], GANs fall short of learning the target distribution and cannot avoid *mode collapse*, i.e., generator only outputs part of the target distribution. Such generated samples would make the training dataset more biased if added, leading to more model bugs. Secondly, even if the generated samples were similar to real world samples, GAN would have difficulty ensuring that the generated samples provide the needed discriminating capabilities for fixing model bugs as *it does not look into the reasons why a NN misbehaves*. Instead, existing works just simply use all generated data or randomly select some samples to fix the bug.

We test 14 different GANs [7, 19, 20, 25, 27] for the MNIST dataset on a trained model which has problems in recognizing digit 5. While the accuracy for the whole dataset is 88%, the accuracy for digit 5 is only 74%. For each GAN, we generate 40,000 new samples as the new training data and use all of them to fix the model bug. The results show that half of the new training sets fail to improve the accuracy for either the model or digit 5, and 4 of them improve the accuracy for the model but not digit 5. Only 3 GANs are able to improve the accuracy for both. However, none of them can improve the accuracy for digit 5 to 83% or higher even after running for 1 hour with all 40,000 new samples, whereas the accuracy can be as high as 94% in 5 minutes using our approach. Note that the original training process finishes within 20 minutes. Such results demonstrate the limitations of existing works.

Figure 2 shows some samples generated by different GANs. The right three are those generated by the three GANs (i.e., CGAN [89], ACGAN [90], and infoGAN [42]) that achieved improvement. Observe that some samples (e.g., the left three) contain substantial noises and are even difficult for humans to recognize. Training with these samples may instead degrade the accuracy of the original model. Another observation is that these samples are very general, do not target on fixing the specific defective behavior (mis-recognizing 6 to 5). However, without understanding why a digit is mis-recognized, the generated images of the digit have limited effectiveness in fixing the problem.

Software Debugging. Our over-arching idea is to learn from the substantial experience of software debugging that is built up by the software engineering community over decades of intensive research (e.g., [31, 36, 47, 52, 53, 67, 81, 92, 95, 111, 113]), and develop new techniques to address the model debugging problem.

As shown in Figure 3, a typical software debugging procedure is as follows. The developer executes the program with a failure inducing input. She then inspects the execution state and compares with the ideal (correct) state that may be derived from her domain knowledge or extracted from similar passing execution(s). We call

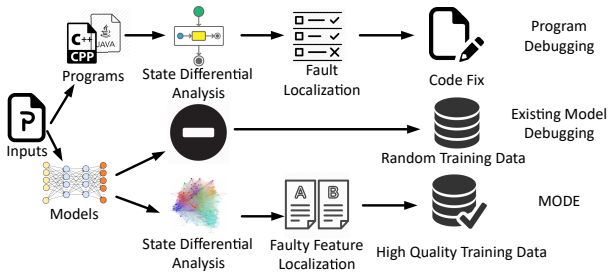
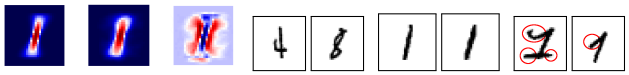


Figure 3: Software Debugging, Model Debugging and MODE

it the *state differential analysis* stage. In this stage, she could use simple methods such as GDB, or advanced techniques such as delta debugging [111] and spectrum based fault localization [70] that compares states of failing execution(s) and passing execution(s). Through differential analysis, the developer locates the root cause and understands the propagation of faulty states. The root cause could be a single-line fault or a more complex fault (e.g., missing functionality). She then fixes the bug by changing the code.

Figure 3 also shows the procedure of existing model debugging, in which the data engineer simply feeds more and more training data, hoping that the misbehavior could be fixed by the new data. As explained before, neural networks are composed of highly complex inter-connected matrices. A misbehavior is usually due to not just one but a large set of neurons. Furthermore, due to the difficulties to understand or interpret neurons [43, 85], it is impossible to directly change weight parameters in a NN like patching source code line(s) in software debugging. Note that a naive approach that simply adds the failure-inducing input (and its clones) to the training set is very problematic [40, 60]. This is because NN works by extracting general features from the training data. Adding many copies of an input to the training set causes the *over-fitting problem* [101], meaning that the extracted features are very specific to some inputs and will have poor performance for general inputs. Analogously in human education, educators cannot fix a student's misunderstanding by asking her to repeatedly work on the same problem. Otherwise, she may just simply memorize the problem and its answer without correcting the underlying misconception.

Our Idea. Inspired by software debugging, we propose a novel neural network model debugging technique that works as follows. As shown in Figure 3, given the inputs related to an output label with buggy behavior (e.g., digit 5 to which many other digits are misclassified), model state differential analysis is performed to identify the features that are critical for the misbehavior (similar to fault localization in software debugging by program state differential analysis). We call these features *faulty features*. Specifically, given the buggy output label and a hidden layer, we develop a light-weight method to determine the importance of each neuron in the hidden layer for the output. We call it a *heat map*. Intuitively, it is an image whose size equals to the number of neurons and the color of a pixel represents the importance of a neuron (for the output). Red denotes the *positive importance* (i.e., the presence of the feature is important for the output) and blue denotes the *negative importance* (i.e., the absence of the feature is important). To facilitate differential analysis, we generate two heat maps for an output label l. One is called the *benign heat map* that is computed from all the inputs that are correctly classified as l and the other the *faulty heat map* computed



(a) Benign (b) Faulty (c) DHM (d) Failing Case (e) Good Sample (f) Bad Sample

Figure 4: Samples and Heat Maps

from all the inputs that are misclassified as l (e.g., images of 4, and 8 in Figure 4(d) that are misclassified as 1). Given a specific bug, such as *under-fitting* of label 1 (the training and testing scores for label 1 are both low), differential analysis is performed to identify the *faulty neurons* that are responsible. For instance, Figure 4(a) and Figure 4(b) show the benign and faulty heat maps of label 1, respectively. Observe that the red areas of the benign heat map roughly show the general shape of the digit, and the blue areas are the regions that are not part of the digit. We find the neurons that are present in the faulty heat map but not in the benign heat map as shown in Figure 4(c), called the *differential heat map*. Intuitively, the highlighted red ones in this differential heat map are the neurons that should be responsible for the misclassification. This is analogous to that in human education, the educator needs to first locate the misconception based on the misbehavior symptom.

After identifying the root cause, the next challenge is how to fix it. As mentioned earlier, a prominent difference between model debugging and software debugging is that one cannot directly modify the faulty neurons. Instead, *we shall reduce their relative influence on the corresponding output label by providing inputs that target training these neurons.* It is a challenge similar to input generation/selection in software regression testing. Specifically, when changes are made to a software, test cases are generated/selected to reach the modified code locations to stress test the new logic. This is achieved by cross-checking the coverage of a test case and the target code locations. Therefore, in model debugging, we generate new inputs using GAN or select inputs from the remaining unused training inputs based on the *differential heat map*. For instance, to fix under-fitting of label 1, we generate/select images of digit 1 that do not contain features in the differential heat map. Intuitively, we are coaching the model to ignore those features (that lead to mistakes) when recognizing 1. Figure 4(e) shows images that have high priority and are more likely to be selected for further training and Figure 4(f) shows images that have low priority and less likely selected. Observe that the latter possesses the faulty features (the circled areas) recognized by our differential analysis. Note that *the selected inputs do not necessarily cause misclassification in the original model.* This is critical as using only misclassified inputs would lead to over-fitting. In our example, after using 500 selected additional images for digit 1 along with 1,500 other random samples in training, we have improved the test accuracy from 81% to 91%, whereas using 40,000 randomly generated images including over 4,000 for digit 1 can only achieve 87%. Simply training on all the misclassified images of digit 1 lowers the test accuracy to 77%. This step is analogous to that in human education, the educator provides additional material/exercises to correct the student's mis-behaviors based on the identified misconception. These material/exercises may not be the ones that the student made mistakes on. Different types of bugs have different kinds of differential analysis. Even for the same kind bug (e.g., under-fitting), we have multiple patching strategies. More details can be found in §3.

3 DESIGN

3.1 Overview

Figure 5 presents the overview of MODE. Given a buggy model, namely, it has over-fitting or under-fitting problems for a label, MODE first performs the model state differential analysis with the correctly classified inputs and misclassified inputs for the label and generates the benign and faulty heat maps for a selected hidden layer (§3.3). The heap maps denote the importance of features for the correct/faulty classification results. As a NN has many hidden layers that represent various levels of abstraction in feature selection, we need to select a layer that likely provides the strongest guidance for fixing the bug. Selecting a layer that is too primitive or too abstract may lead to sub-optimal results. In particular, the features in a primitive layer (close to the input layer) may be too general (contributing to many output labels). Retraining them may not have unique effect on the target label. On the other hand, the features in a very abstract layer (close to the output layer) may be too specific so that it is difficult to identify the faulty features, e.g., each feature may abstract part of the faulty behaviors in the earlier layers and contribute a little bit to the misclassification but none of them is dominating. We develop an algorithm to select the layer that strikes a balance between the two depending on the bug type (Algorithm 1). Based on the generated benign and faulty heat maps for the selected layer, MODE performs various differential analyses (according to the bug type) to generate the differential heat map, which highlights the target features for retraining. The differential heat map is used as guidance to select existing or new inputs (generated by GANs or collected from the real world) to form a new training dataset, which is then used to retrain the model and fix the bug (§3.4).

3.2 Model Bug Types and Fixes

As discussed in §2, in this paper a model bug refers to that the test accuracy of a specific label in the model (with a fixed structure) is lower than the best possible accuracy. Note that the best possible accuracy is usually not 100% due to the inherent uncertainty of the application. Since the ideal accuracy is unknown, in practice we consider a model is buggy if its test accuracy can be improved.

For a given model M with a set L of all possible labels, each input can be tagged with a tuple (g, p) where g is the groundtruth label and p is the predicted label. Based on these tuples, the test results can be classified to two categories: the correctly predicted inputs ($p = g$) and mis-predicted inputs ($p \neq g$). We use $S^I_{g,p}$ to represent the input subset whose groundtruth label is g and the predicted label is p with the input dataset I. Standard machine learning process involves at least two sets of data: the training dataset T, and the test dataset D which is unknown while training. Sometimes it involves one more validation set. To measure the performance of the model, we define a few terms.

$$TrAcc(M^T_g) = \frac{|S^T_{g,g}|}{\Sigma|S^T_{g,*}|}, \quad TeAcc(M^D_g) = \frac{|S^D_{g,g}|}{\Sigma|S^D_{g,*}|}$$

$TrAcc(M^T_g)$ calculates the percentage of correctly predicted cases for label g on the training dataset (known as the *training accuracy for label g*), and $TeAcc(M^D_g)$ measures that on the test dataset (known as the *test accuracy for label g*).

Under-fitting bugs. Since the best possible accuracy is unknown, in practice we say that a model has an *under-fitting bug*, if

$$\exists g, TrAcc(M^T_g) \leq \theta, TeAcc(M^D_g) \leq \theta$$

where θ is a pre-defined value based on concrete applications. Such values can be derived from the statistics of similar models in the wild. For image processing applications such as face recognition, this value can be relatively high (e.g., 90%) as neural networks are good at such tasks. And for many other applications, e.g., natural language processing (NLP), the value is relatively lower (e.g., 60%). *Under-fitting bugs mean that the model can neither properly model the training data nor generalize to new data.* This is usually because the unique features for this category are not appropriately extracted, or the connections between the unique features to the output label are too weak. From the training perspective, this is because the training samples are too diverse, containing many *noisy/harmful* training samples [59]. To fix the problem, we need to provide enough high quality new training inputs. MODE aims to improve the quality of selected input samples. High quality samples contain a lot of label-unique features (e.g., Figure 4(e)) while noisy/harmful training samples introduce a lot confusing features to the model (e.g., Figure 4(f)), which are responsible for many mis-predicated cases (e.g., Figure 4(d)). Take the circled red areas in Figure 4(f) as an example. They will not help improve model precision, but rather induce mis-classification.

Over-fitting bugs. We say that a model has an *over-fitting bug*, if

$$\exists g, TrAcc(M^T_g) - TeAcc(M^D_g) > \gamma$$

where γ is a pre-defined value based on concrete use scenarios. The rule means that the training accuracy for label g is much higher than the test accuracy. This indicates that M can model existing known data very well, but cannot generalize to new data. In other words, the model is over-trained on some features. To fix the problem, we ought to add samples to the training dataset to downplay such features. Figure 6 shows an over-fitting example for digit 7. The heat map representation and one example is shown in Figure 6(a). As we can see, the model is over-fitted to a specific handwritten style that has a cross-bar. As a comparison, the heat map of a well trained model with more different handwritten styles of digit 7 and one example style is shown in Figure 6(b).

3.3 Model State Differential Analysis

The model state differential analysis is mainly to help data engineers understand bugs in the model, that is, what features are important for a bug. In particular, it includes (1) selecting the layer whose features are likely to provide the strongest guidance in bug fixing; (2) analyzing the importance of the features in the layer for correct/faulty classification and generating the heat maps.

Layer Selection. As discussed in §2, neural network models extract features through the hidden layers. A hidden layer closer to the output layer contains features that are more abstract. The goal of layer selection is to find the layer that is most susceptible for improvement. The key observation is that the level of abstraction denoted by different layers have various effects for model accuracy. Figure 7(a) and Figure 7(b) show the typical effect (test accuracy) of using different number of layers. The solid lines show the accuracy

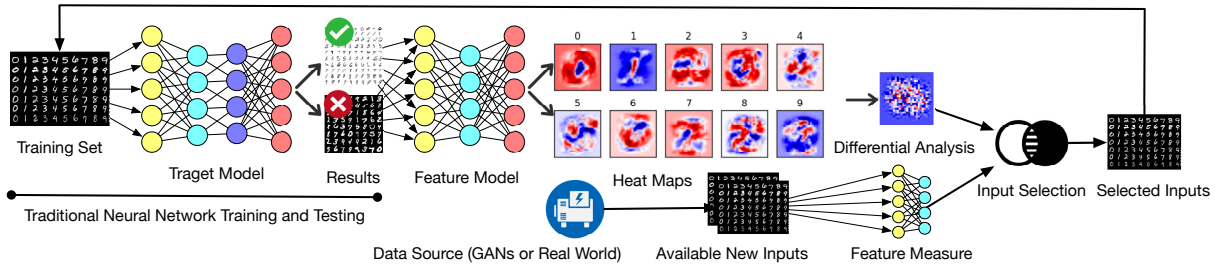
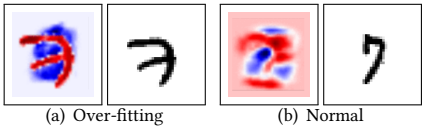
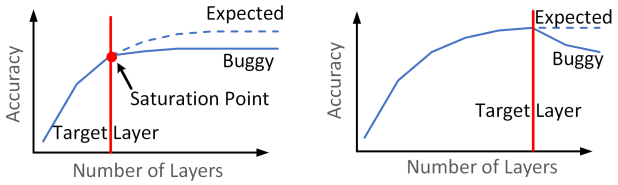


Figure 5: Overview of MODE Using MNIST as an Example



(a) Over-fitting (b) Normal

Figure 6: Over-fitting and Normal Cases for 7



(a) Under-fitting Bug (b) Over-fitting Bug

Figure 7: Test Accuracy v.s. Number of Layers

change along with the number of layers. At the beginning, having more layers leads to significant accuracy improvement because more abstract features are discovered. At a certain point, called the *saturation point* as shown in the figure, the accuracy reaches the proximity of the peak value and the line becomes flat. This is usually because the extracted features are good enough (general as well as unique for a specific task) and adding more hidden layers can hardly improve the accuracy. At the tail of the line, as shown in Figure 7(b), we may observe that adding too many layers leads to accuracy degradation as the features become too specific to the given training data, and the neural network is just remembering the training data without generalizing (due to the larger capacity provided by the additional layers), causing over-fitting. Such situations have been well studied by previous research [61, 62].

It is important for MODE to choose a proper layer for the further analysis, we call it the *target layer*. In particular, MODE chooses the layer that the accuracy reaches or leaves the saturation point (e.g., the layer denoted by the red lines in Figure 7), indicating the edge to advance (e.g., the dotted lines represent the expected performance after bug fixing). That is, MODE chooses the first layer that has a limited test accuracy difference with its previously analyzed layer. Intuitively, analyzing features in this layer would give us the most prominent evidence of the bug as it denotes the turning point of accuracy improvement. We have slightly different methods to choose the target layer based on the bug type. For under-fitting bugs, MODE performs the *forward analysis* with the direction from the input layer to the output layer, whereas for over-fitting bugs, MODE performs the *backward analysis* with the direction from the output layer to the input layer. Intuitively, for under-fitting, selecting the forward saturation point allows us to identify the layer where abstraction is about to saturate so that we can add more inputs to enrich the features. In contrast, selecting the backward saturation point allows us to identify the place that the model is about to become over-abstracted so that we can add more inputs to suppress faulty features.

Algorithm 1 Layer Selection for Under-fitting

```
1:  function SELECTLAYER(model M, DataSet DS)
2:      i ← M.inputLayer()
3:      o ← M.outputLayer()
4:      lastdis ← init()
5:      for each hidden layer l from i towards o do
6:          m ← M.subModel(i, l)
7:          m.freeze()
8:          lfm ← m.add(o.copyStructure())
9:          lfm.train(DS.trainingSet())
10:         ldis ← lfm.test(DS.testSet())
11:         lsimscore ← BhattacharyyaDistance(lastdis, ldis)
12:         if lsimscore ≤ α then
13:             return M.layerBefore(l)
14:         else
15:             lastdis ← ldis
16:     return o
```

The algorithm of selecting the target layer for fixing an under-fitting bug is presented in Algorithm 1. It loops over each hidden layer in the forward fashion. For each layer l, it extracts a sub-model which contains all the layers up to l and then freezes the weight parameters in the corresponding matrices so that the later training would not change their values. Then a new output layer is added that has the same output labels as the original model to construct a new model, and this model is known as the *feature model* (Figure 8). We retrain the feature model with the same training data. Notice that only the last layer is updated during retraining and hence the training efforts are small (i.e., seconds to a few minutes). Intuitively, we are trying to use part of the original model (and hence the features abstracted by the submodel) to make predictions. It then tests the trained feature model and compares with the result by the previous feature model using the Bhattacharyya Distance (line 11), which is widely used to measure the similarity between probability distributions [32, 58, 93]. If they are very similar, the layer before l is considered the target layer. The algorithm for over-fitting is similar and hence omitted.

Measuring Feature Importance. After identifying the target layer, the next step is to measure the importance of individual features (or neurons) in this layer for correct and faulty output results. Note that although the NN training process inherently computes certain sensitivity information, called *gradients*, which predict how much output changes may be caused by certain weight parameter

changes. However, *importance is different from sensitivity*. The former is global and with respect to features, whereas the latter is local and with respect to weight parameters (i.e., the connections between features across layers). Specifically, sensitivity measures that given the current weight value, how much output change can be induced by a (small) Δ of the weight value. In contrast, importance measures how much influence does a feature have on the classification result of an output label. An important feature may not have sensitive weights. For example, the red region in Figure 4(a) denotes the features that are important for recognizing digit 1. But their parameters may not have large sensitivity, that is, small changes to the parameters can hardly impact the classification results.

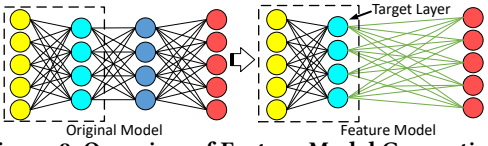


Figure 8: Overview of Feature Model Generation

To measure importance, our idea is to leverage the feature model. Recall that the feature model reuses part of the feature selection layers in the original model (from the input layer up to the target layer) and retrains the output layer, which can be considered making predictions based on the features in the target layer (e.g., through the *softmax* function). The boxed area in Figure 8 represents the selected sub-model. As shown on the right-hand-side, the feature model directly connects the target layer with a new output layer. The green lines in the feature model, connecting the target layer and the output layer, are retrained. *After retraining, the weight values of the connections from the individual feature/neurons in the target layer to a label in the output layer essentially denote the importance of the features for the output label.* Specifically, we use the matrix representing the connections from the target layer to the output layer to construct a heat map, by normalizing the weights to the range $[-1, 1]$. The absolute value of a weight hence represents the importance of the corresponding feature. For example in Figure 8, the green lines are retrained, and represent the importance of individual features for output labels. The values are hence normalized and used for generating the heat map. Images in Figure 4 and Figure 6 visualize feature importance. The red color represents the positive values ($(0, 1]$), and the blue color represents the negative values ($[-1, 0)$). Features of no or little importance (close to 0) are in white. The number of heat maps equals to the number of output labels, and the size of each heat map equals to the size of features (neurons). Formally, we use $H_l[i]$ and w_l^i to represent the weight value of the ith feature for the label l, and a heat map for label l, H_l can be represented as:

$$H_l = [w_l^0, w_l^1, \ldots, w_l^n]$$

3.4 Differential Heat Map and Input Selection

Differential heat map identifies the features that are critical for the faulty behavior, called the *faulty features*. This is achieved by contrasting heat maps for correctly classified and mis-classified inputs. Specifically, given an output label l that we aim to debug, there are two groups of inputs: CI_l represents correctly predicated inputs for l, and MI_l and WI_l containing mis-predicated inputs with

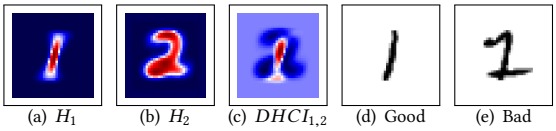


Figure 9: DHCI for Digits 1 and 2

MI_l denoting cases that are misclassified as l, and WI_l denoting cases of l that are misclassified as others. MODE generates a heat map for each group, HCI_l for CI_l, HMI_l for MI_l and HWI_l for WI_l.

Under-fitting Bugs. To understand and patch an under-fitting bug of label l, we need to provide more inputs of label l with features that are unique for l and suppress the faulty features (causing misprediction). To enhance unique features for label l, we perform a differential analysis on the correctly predicated dataset CI to get the differential heat map $DHCI$, which is computed as follows.

$$DHCI_{l,k}[i] = HCI_l[i] - HCI_k[i]$$

$$DHCI_l[i] = DHCI_{l,k}[i], \text{ with } k \neq l, k \in L, \; abs(DHCI_{l,k}[i]) \text{ is min}$$

$DHCI_{l,k}[i]$ computes the importance difference of the feature i between the label l and k, and $DHCI_l[i]$ records the smallest difference. If the value is very small, it means feature i is equally important for both l and another feature and hence not so unique. Otherwise, the feature is very unique to l. Figure 9 shows an example of computing $DHCI_{1,2}$ for the MNIST dataset. In Figure 9(c) (the differential heat map), the red color represents the uniquely important features for digit 1 which are not in digit 2, and the blue color denotes the features that are unique for 2 but not for digit 1. As we can see, the shared important features are now white as the distance is almost 0. Thus to enhance the unique features, we ought to select samples with strong presence of features in the red areas (e.g., Figure 9(d)), and lower the priority of selecting the samples with strong presence of features in the blue areas (e.g., Figure 9(e)).

To suppress faulty features for label l, we perform differential analysis on the mis-predicted dataset MI_l (those misclassified as l) and CI_l to acquire a heat map $DHMI_l$ as follows:

$$DHMI_l[i] = HMI_l[i] - HCI_l[i]$$

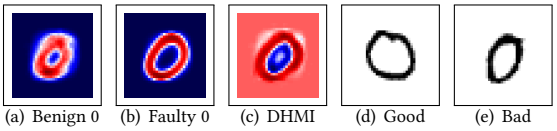
$DHMI_l[i]$ denotes the importance difference of the feature i for faulty and correct classifications of l. A small absolute value means the feature i leads to similar behaviors in both categories and is hence not very relevant. A large positive value (red) indicates that i is particularly important for misclassification and hence faulty. An example is shown in Figure 4 (Figure 4(a) (HCI_1), Figure 4(b) (HMI_1) and Figure 4(c) ($DHMI_1$)).

Over-fitting Bugs. As over-fitting bugs are essentially caused by biased training samples whose features are too specific, we need more samples especially the ones with more diversity to fix the bug. The failing samples for the over-fitted label tells the most needed features. Thus we perform a differential analysis on these inputs, which computes a differential heat map $DHWI_l$ as follows:

$$MHWI_l[i] = HWI_{l,k}[i], \text{ with } k \neq l, k \in L, abs(HWI_{l,k}[i]) \text{ is max}$$

$$DHWI_l[i] = MHWI_l[i] - HCI_l[i]$$

$HWI_{l,k}$ denotes the heat map for the inputs of l that are misclassified as label k. $MHWI_l[i]$ computes the maximal importance value that feature i has for some misclassification. Hence large values



(a) Benign 0 (b) Faulty 0 (c) DHMI (d) Good (e) Bad

Figure 10: Differential Heat Map for Over-fitted 0

(a) FM (b) CIFAR (c) OD (d) AC

Figure 11: Samples for Evaluated Application

(red areas) in $DHMI_I$ represent the features that are unique for the misclassified cases and currently ignored by the model, and hence potentially important for generalization. As a result, we ought to select samples with strong presence of such features.

Figure 10 shows an over-fitting bug. Figure 10(a) shows the heat map of the model for digit 0 (HCI_0), and Figure 10(b) shows the heat map for cases of 0 that are misclassified as other (HWI_0). The generated differential heat map, $DHWI$ is Figure 10(c). Its red areas denote the potential new features for generalizing the model, and its blue color areas denote the existing over-fitted features. As we can see, in this case, the model was trained on small sized digit 0 images, and hence large sized digit 0 are mis-predicted to other digits (e.g., 8). The differential heat map provides clear suggestion that larger digit 0 images shall be used.

Input Selection. With the generated differential heat maps, we can perform input selection from the available new inputs. For each new input s, we first feed it to the feature model (without running through the output layer) to acquire a feature value vector v_s. For a given input and a differential heat map DH, we calculate a score rs_I by computing the dot product, $rs_I = v_s \cdot DH$. Intuitively, the differential heat map is a vector pointing to the most effective direction to fix the bug, and thus the score measures the contribution of the input along the direction. Note that MODE does not exclusively use new samples based on the scores as we do not want to overfit for the buggy label. Hence, we also use additional random samples. In this paper, the ratio between selected and random samples is 1 : 3. Our experiments show that adding too many high scored samples may degrade the final accuracy.

4 EVALUATION

We implement a prototype on TensorFlow [28]. In the evaluation, we aim to address the following research questions:

RQ1: How effective and efficient is MODE in fixing model bugs?
RQ2: How does MODE compare to using random samples or faulty samples to fix model bugs?
RQ3: What is the impact of different parameters?

4.1 Fixing Model Bugs

To answer RQ1 and RQ2, we collect multiple models for a few applications. These models may be trained on the same dataset with different hyper-parameters, model structures (e.g., number of layers), and activation/loss functions etc. The models have various sizes, ranging from 7k parameters (e.g., [24]) to over 20M parameters (e.g., [4]). Part of the models and their stats are shown in Table 1.

• **MNIST**: We use the MNIST handwritten digit dataset (60,000 training and 10,000 testing samples), and collect 9 different NN models from Github published by various groups such as Google [24].

• **Fashion MNIST (FM)**: We use the Fashion MNIST dataset [15, 109] (60,000 training and 10,000 testing samples), and 6 models [11, 16–18, 22, 30] published on its web page. Figure 11(a) shows an example of the application.

• **CIFAR-10**: For object recognition, we use the CIFAR-10 dataset [78] (50,000 training and 10,000 testing samples), which contains different images for 10 types of objects with equal number of training and testing samples per type. We use 5 different classifier implementations from Github [4]. Examples are shown in Figure 11(b).

We split the original dataset into four parts: training set (30%), validation set (10%), test set (10%) and bug fixing set (50%). The bug fixing set is reserved for fixing the buggy model, the training set is used for training only, the validation set is used for validating the trained model during training and the test set is assumed unknown during training and (only) used to test the performance of the trained model. In order to compare with existing data augmentation/generation solutions, we collect a large number of GAN implementations as well. For MNIST, we find 14 different GAN implementations [7, 19, 20, 25, 27]; for FM, we collect 10 different GAN implementations from [20]; for CIFAR, we find three GAN implementations [6, 8, 9]. Also, we compare MODE with a naive approach that reuses the mis-classified data as new inputs.

For each model, we select an under-fitting bug, i.e., the output label with the lowest training and testing accuracy, and an over-fitting bug, i.e. the label with good training accuracy but the lowest test accuracy. The retraining proceeds in batches, each having 2,000 new samples. After each batch, we evaluate the updated model on the test dataset. If the test accuracy for both the whole model and for the specific buggy label become higher, and the accuracy of the label is no longer substantially lower than the model accuracy, we consider the bug is fixed. Otherwise, we continue training with more batches. We set a limit of 20,000 samples or 2 hours for 5 MNIST models and all Fashion MNIST models (MNIST-1 to MNIST-5, and FM-1 to FM-6 in Table 1), and 40,000 samples or 4 hours for the remaining 4 MNIST models (MNIST-6 to MNIST-9) and 3 CIFAR models (CIFAR-1 to CIFAR-3 in Table 1). For the other two CIFAR models (CIFAR-4 and CIFAR-5), as they are extremely large with over 20 million parameters to train for each model, we set the constraint to be 40,000 samples or 24 hours. If the retraining is not able to finish fixing a bug within the limit (according to the aforementioned standard), we report the results upon termination.

Table 1 shows the experimental results including (left to right) the model and its size measured by the number of parameters to train, the bug type, and the accuracy for both the model and the specific buggy label. For MODE, we show the number of samples used to fix the bug (selected+random samples), the time used for retraining, the test accuracy for the model and the buggy label. For the GAN approach, we show the number of GAN models that improve the test accuracy by 5+%, and the highest test accuracy for the model and the buggy label, the number of samples used to train this model and the training time. The last two columns present the results of using all the mis-predicated samples to fix bugs.

Table 1: Fixing Model Bugs Summary

Model (Size)	Bug Type	MAcc	LAcc	MODE					Randomly Selecting GAN				Failing	
				#S	T	MAcc	LAcc	>5%?	MAcc	LAcc	#S	T	MAcc	LAcc
MNIST-1	U	88%	74%	500+1500	5m	93%	94%	3(14)	90%	83%	20000	1h2m	86%	80%
7k	O	84%	81%	500+1500	6m	92%	91%	4(14)	89%	87%	20000	1h4m	85%	77%
MNIST-2	U	89%	72%	500+1500	5m	92%	92%	5(14)	90%	88%	20000	57m	85%	82%
185k	O	84%	76%	500+1500	5m	93%	93%	3(14)	87%	85%	20000	1h4m	83%	74%
MNIST-3	U	86%	76%	500+1500	5m	94%	94%	4(14)	92%	87%	20000	1h10m	88%	82%
185k	O	84%	78%	500+1500	5m	94%	95%	3(14)	93%	90%	20000	55m	84%	82%
MNIST-4	U	86%	78%	500+1500	5m	94%	94%	3(14)	86%	84%	20000	56m	85%	80%
185k	O	84%	74%	500+1500	5m	92%	92%	4(14)	84%	83%	20000	50m	88%	78%
MNIST-5	U	82%	77%	500+1500	5m	94%	92%	4(14)	87%	88%	20000	54m	84%	74%
122k	O	85%	77%	500+1500	5m	92%	92%	3(14)	88%	90%	20000	54m	82%	76%
MNIST-6	U	84%	72%	1000+3000	10m	93%	93%	3(14)	89%	84%	40000	1h58m	81%	78%
244k	O	84%	74%	1000+3000	9m	94%	94%	3(14)	86%	82%	40000	2h	79%	76%
MNIST-7	U	87%	77%	1000+3000	9m	93%	93%	3(14)	88%	85%	40000	2h9m	84%	75%
185k	O	85%	72%	1000+3000	9m	93%	91%	3(14)	87%	88%	40000	2h4m	87%	82%
MNIST-8	U	86%	73%	1000+3000	10m	93%	93%	3(14)	87%	82%	40000	1h54m	88%	76%
185k	O	84%	73%	1000+3000	12m	93%	94%	3(14)	88%	85%	40000	2h6m	87%	76%
MNIST-9	U	84%	73%	1000+3000	9m	94%	95%	3(14)	88%	88%	40000	2h	81%	73%
257k	O	84%	72%	1000+3000	9m	93%	93%	3(14)	86%	86%	40000	2h3m	87%	77%
FM-1	U	88%	80%	500+1500	5m	93%	90%	2(10)	84%	88%	20000	1h2m	83%	78%
493k	O	87%	82%	500+1500	5m	94%	94%	3(10)	89%	90%	20000	1h4m	88%	84%
FM-2	U	85%	77%	500+1500	5m	95%	95%	2(10)	89%	88%	20000	1h9m	87%	80%
1.2M	O	87%	74%	500+1500	5m	94%	94%	2(10)	90%	84%	20000	1h3m	85%	80%
FM-3	U	87%	72%	500+1500	10m	93%	91%	2(10)	89%	78%	20000	1h12m	89%	76%
3.2M	O	85%	69%	500+1500	9m	93%	93%	2(10)	88%	88%	20000	1h7m	88%	78%
FM-4	U	86%	73%	500+1500	5m	92%	94%	3(10)	83%	80%	20000	1h3m	87%	73%
765k	O	85%	74%	500+1500	5m	91%	92%	1(10)	88%	80%	20000	1h9m	87%	75%
FM-5	U	87%	80%	500+1500	5m	92%	92%	2(10)	87%	86%	20000	1h3m	79%	74%
113k	O	83%	73%	500+1500	5m	92%	93%	2(10)	83%	82%	20000	1h	86%	80%
FM-6	U	89%	81%	500+1500	5m	93%	95%	2(10)	91%	91%	20000	1h3m	83%	75%
26M	O	82%	74%	500+1500	9m	92%	94%	3(10)	85%	83%	20000	1h2m	85%	80%
	Avg	85%	75%		7m	93%	93%		88%	85%		78m	85%	78%
CIFAR-1	U	79%	64%	500+1500	6m	88%	89%	0(3)	74%	64%	40000	1h14m	79%	66%
62k	O	79%	65%	500+1500	7m	92%	91%	0(3)	82%	63%	40000	1h21m	80%	68%
CIFAR-2	U	84%	76%	500+1500	15m	91%	90%	0(3)	80%	74%	40000	2h40m	81%	82%
0.97M	O	83%	72%	500+1500	21m	88%	89%	0(3)	88%	79%	40000	2h50m	85%	78%
CIFAR-3	U	82%	78%	500+1500	30m	91%	90%	0(3)	81%	78%	40000	4h10m	83%	74%
1.7M	O	86%	83%	500+1500	24m	93%	92%	0(3)	84%	74%	40000	4h9m	80%	72%
CIFAR-4	U	84%	74%	1000+3000	12h40m	92%	93%	0(3)	87%	75%	38000	24h	86%	74%
20M	O	87%	78%	1000+3000	12h9m	91%	92%	0(3)	90%	77%	38000	24h	87%	77%
CIFAR-5	U	88%	79%	1000+3000	10h	92%	94%	0(3)	85%	78%	40000	24h	88%	78%
20M	O	86%	79%	1000+3000	9h40m	93%	94%	0(3)	88%	78%	40000	24h	86%	73%
	Avg	84%	75%			91%	91%		84%	74%			84%	74%

From Table 1, we make a few observations. Firstly, compared with the other two approaches, MODE is more effective (higher test accuracy) and efficient (less training time and fewer samples) in fixing the model bugs. In the meantime, the model test accuracy is also improved after MODE fixing the bugs, indicating that MODE does not degrade the effectiveness of the whole model. Thus, we can say that MODE can effectively and efficiently fix model bugs without introducing new bugs. Secondly, randomly selecting GAN generated inputs can fix some bugs, but fails on many others. Also, it can potentially introduce new bugs, causing the degradation of model accuracy. Even in cases where this approach can fix bugs, it requires 11x longer training time and more data samples, leading to larger overhead. Figure 12 shows the test accuracy change along with the number of new samples for model MNIST-1 during retraining. As we can see, MODE can improve the accuracy quickly because of the high quality data samples. Using random selected GAN samples can sometimes improve the accuracy, but in many other cases, it may degrade the accuracy as ineffective/bad samples are chosen. We consider the feature of quick improvement is very important when the retraining budget is limited. Furthermore, directly re-using mis-predicted samples as the new training samples to fix bugs, in most cases, leads to the over-fitting problem, causing the degradation of test accuracy. Another observation we have is that even though the buggy model is trained with less training data, the root cause of the bug is NOT because of lacking training

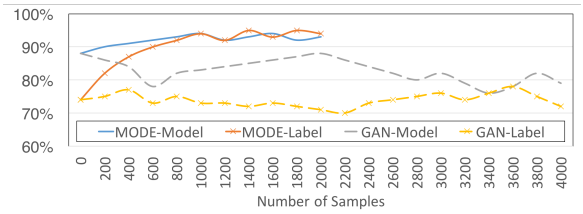


Figure 12: Test Accuracy vs. Training Samples

Table 2: Accuracy Improvement without GANs

Model	Bug	Original		MODE		Random	
		MAcc	LAcc	MAcc	LAcc	MAcc	Lacc
FR	OF	76%	65%	88%	84%	79%	72%
2.1M	UF	72%	64%	85%	86%	78%	70%
OD	OF	83%	74%	89%	88%	84%	77%
3.2M	UF	82%	75%	88%	83%	84%	79%
AC	OF	33%/44%	13%/22%	46%/60%	38%/47%	32%/40%	33%/42%
30M	UF	25%/36%	11%/20%	42%/52%	36%/44%	32%/41%	25%/32%

data. Notice that even after MODE fixes the bug with new training data, the total training data is still smaller than the original dataset. However, many models actually achieves better performance than training with the original dataset. For example, the MNIST-1 model achieves 92% test accuracy on the 60,000 training samples while MODE achieves 94% with 30% of this plus 2000 new samples.

Applications without Inputs by GANs. We also evaluate RQ1 and RQ2 for applications and models without available GANs to generate new inputs. We divided the dataset into 4 parts as described before. Given a model bug, we use MODE and the random selection approach to select the same number of inputs (from the reserved dataset) as the new training data to fix the bug, and measure the improvement of each method. Here, we select 1,000 inputs to retrain. The retraining time ranges from 5 minutes to 30 minutes. Due to the space limit, we only use one model for each application:

- **Face Recognition**(FR): We use the Labeled Faces in the Wild (LFW) dataset [66] (10000 training and 3233 testing samples), and the model is directly obtained from Github [14].
- **Object Detection**(OD): This model [3] tries to detect the existence of eyeglasses. The images are from the CelebA dataset [86] (162770 training, 19867 validation and 19962 testing samples).
- **Age Classification**(AC): This classifier [21] predicts the age group of a person based on the dataset from the OUI-Adience Face Image Project [50] (15,000 training and 2,523 testing samples). The ages are divided into 8 groups. As this is a very difficult problem even with human intelligence, the model is measured not only by accuracy but also by off-one accuracy, meaning that a prediction result is considered correct if it is the ground truth or a one-step neighbor of the ground truth.

The experimental results are shown in Table 2. For AC, both accuracy and one-step accuracy are reported. Observe that for these applications, MODE is still highly effective in improving the accuracy for over-fitting (OF) or under-fitting (UF) bugs for both the model and the buggy label. Randomly selected inputs are helpful in general. But they are less effective. Sometimes, they lead to accuracy degradation (e.g., the over-fitting bug for the age classification).

Debugging Pre-trained Models. Another experiment we did is to apply MODE on *models that have relatively high accuracy and do not have obvious bugs in output classes.* We use an iterative method to fix the bugs. During each iteration, we first identify the most buggy

Table 3: Real-world Models Bug Fix

DataSet	Model	Original Acc.	# Samples	MODE Acc.	Random Acc.
MNIST	MNIST-10 [23]	95.2%	2000	97.4%	94.8%
	MNIST-11 [23]	93.4%	2000	96.8%	94.3%
Fashion MNIST	FM-7 [15]	87.6%	2000	92.3%	88.9%
	FM-8 [15]	91.6%	2000	92.6%	88.5%
CIFAR	CIFAR-6 [5]	87.3%	4000	93.2%	87.3%
	CIFAR-7 [5]	88.4%	4000	92.8%	88.2%

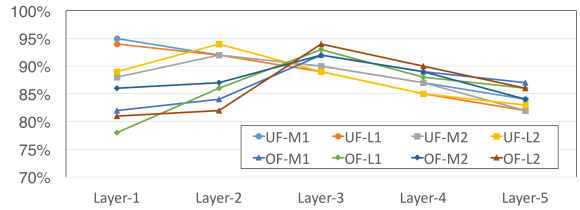


Figure 13: Effects of Selecting Different Target Layers

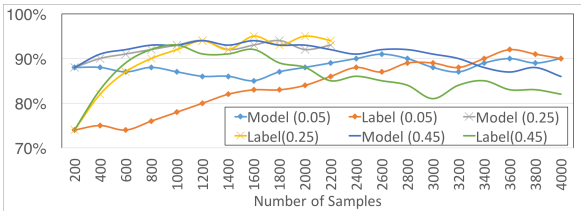


Figure 14: Effects of Using Different α

output class by calculating the difference between the training accuracy and validation accuracy, and use MODE to fix the bug. The procedure continues until the accuracies for each output class are above a threshold or the model accuracy cannot be improved.

We use 5 real-world pre-trained models [15, 23, 78] that are trained on the original dataset with relatively high test accuracy and no obvious bugs for all output classes. To fix these model bugs, we use collected GANs to generate the validation dataset and bug fix datasets and compare MODE with the random input selection approach. Table 3 summaries the results. For each model, we show the original model accuracy (column 4), the number of new samples used to fix the model for both approaches (column 5), the new accuracy using MODE (column 6) and the average accuracy of using the random approach 10 times (column 7). From the table, we can clearly see that MODE is more effective and efficient than using the random approach in all cases.

4.2 Design Choices

To answer RQ3, we perform a number of experiments using different parameters to see how this affects the performance of MODE.

Effects of Layer Selection. In §3, we discussed choosing different layers (to generate heat maps) may affect the effectiveness of MODE. In this experiment, we study such effects. Figure 13 shows 4 cases: 2 under-fitting bugs and 2 over-fitting bugs on 4 different models (MNIST-2 to MNIST-5). All these four models have 5 layers. As we can see, there is always an optimal layer that leads to the best test accuracy. Also, our proposed method Algorithm 1 is able to select the optimal layer for all cases. This demonstrates the effectiveness of layer selection. The experiments on a few other applications and models demonstrate the same result. Details are elided.

Ratio between Selected Data and Random Data Another important parameter in MODE is the percentage α of selected data samples in the new training dataset. Due to space limitations, we only show the results of the MNIST-1 model, and the conclusion is applicable for other models. Figure 14 shows the test accuracy change for both the model and the buggy label during training using three different α values: 0.05, 0.25 and 0.45. When the value is very small (0.05), MODE can still fix the model, but requires a lot more training samples and time. When using a proper value, in this case, $\alpha = 0.25$, MODE can fix the model bugs with the minimal training efforts. When the value is too large, such as $\alpha = 0.45$, MODE can quickly improve the test accuracy for the buggy label, but in the meantime easily triggers over-fitting, leading to degradation of test accuracy for both the model and the buggy label.

5 RELATED WORK

Our work is inspired by software engineering techniques, especially software debugging and regression testing. Software debugging has been extensively studied, producing a large number of highly effective techniques (see §2). Many of these techniques work by performing differential analysis on program execution states (e.g., comparing variable values in passing and failing runs). The success of these techniques inspires us to contrast NN model states. However, different from software debugging, model bugs cannot be fixed by directly changing model parameters, but rather through retraining. The inter-dependencies of model internal states are much more complex compared to program dependencies, requiring different solutions. In regression testing, how to efficiently select/prioritize test cases is an important challenge. There are also many highly effective solutions [35, 48, 51, 55, 69, 83, 84, 91, 98, 100, 104, 110] The basic idea is to check the correlations between individual test cases and code modifications. This inspires us to select inputs for model debugging by comparing the strong features possessed by individual inputs and the features that we target to modify.

Machine learning techniques are widely used in various software engineering applications [41, 44, 72–74, 82, 103, 106–108]. MODE has the potential to facilitate researchers to debug their models. In recent years, researchers proposed various methods to address the machine learning model debugging problem [38, 39, 76, 114]. However, these techniques are limited to specific machine learning model types, and cannot handle complex models like neural networks. Furthermore, they do not perform differential analysis to identify faulty features before fixing them. In [68, 114], researchers aim to identify incorrect items in the training set and clean up training data. Here, incorrect data means corrupted data (e.g., ill-formatted XML files). These work are orthogonal to MODE. There are also works [76, 77, 80, 87] aiming at providing provenance information and explanation of models to data engineers to help them understand or debug the models. They require human inspection, while MODE is an automated technique.

6 CONCLUSION

Inspired by software debugging and regression testing techniques, we propose and develop MODE, an automated neural network debugging technique powered by state differential analysis and input selection. It can help identify buggy neurons and measure their importance to guide the new input sample selection. MODE can construct high quality training datasets that effectively and efficiently fix model bugs without introducing new bugs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable comments. This research was supported, in part, by the United States Air Force and DARPA under contract FA8650-15-C-7562, NSF under awards 1748764, 1409668, and 1320444, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2018. *Alexa, Nein!* https://www.telegraph.co.uk/news/2017/11/08/alexa-nein-police-break-german-mans-house-music-device-held/ (Accessed on 02/28/2018).
[2] 2018. *Caffe Model Zoo.* https://github.com/BVLC/caffe/wiki/Model-Zoo.
[3] 2018. *CelebA.* https://github.com/swatishr/CNN-on-CelebA-DataSet-using-Tensorflow.
[4] 2018. *Cifar 10 CNN.* https://github.com/BIGBALLON/cifar-10-cnn.
[5] 2018. Classification datasets results. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. (Accessed on 07/24/2018).
[6] 2018. *CPPN GAN Cifar.* https://github.com/hardmaru/cppn-gan-vae-cifar-tensorflow.
[7] 2018. *DCGAN.* https://github.com/znxlwm/tensorflow-MNIST-GAN-DCGAN.
[8] 2018. *DCGAN.* https://github.com/shaohua0116/DCGAN-Tensorflow.
[9] 2018. *DCGAN CIFAR10.* https://github.com/4thgen/DCGAN-CIFAR10.
[10] 2018. dnnmode/buggymodels. https://github.com/dnnmode/buggymodels.
[11] 2018. *F-MNIST Keras.* https://github.com/QuantumLiu/fashion-mnist-demo-by-Keras.
[12] 2018. *Face ID Broken.* https://www.wired.com/story/hackers-say-broke-face-id-security/.
[13] 2018. *Face Recognition.* https://pypi.python.org/pypi/face_recognition.
[14] 2018. *Face Recognition.* https://github.com/seathiefwang/FaceRecognition-tensorflow.
[15] 2018. *Fashion MNIST.* https://github.com/zalandoresearch/fashion-mnist.
[16] 2018. *Fashion MNIST.* https://github.com/Xfan1025/Fashion-MNIST/blob/master/fashion-mnist.ipynb.
[17] 2018. *Fashion MNIST CNN.* https://github.com/abelusha/MNIST-Fashion-CNN/blob/master/Fashon_MNIST_CNN_using_Keras_10_Runs.ipynb.
[18] 2018. *Fashion MNIST convnet.* https://github.com/zalandoresearch/fashion-mnist/blob/master/benchmark/convnet.py.
[19] 2018. *GAN-MNIST.* https://github.com/yihui-he/GAN-MNIST.
[20] 2018. *GANs.* https://github.com/hwalsuklee/tensorflow-generative-model-collections.
[21] 2018. *Gender Age.* https://github.com/naritapandhe/Gender-Age-Classification-CNN.
[22] 2018. *kashif NN.* https://gist.github.com/kashif/76792939dd6f473b7404474989cb62a8.
[23] 2018. *MNIST.* http://yann.lecun.com/exdb/mnist/.
[24] 2018. *mnist tutorial.* https://github.com/martin-gorner/tensorflow-mnist-tutorial.
[25] 2018. *RGAN.* https://github.com/ratschlab/RGAN.
[26] 2018. *Self-Driving Bus Crashes Launch.* http://cbslocal.com/2017/11/08/self-driving-shuttle-bus-crashes-las-vegas/.
[27] 2018. *SSGAN.* https://github.com/gitlimlab/SSGAN-Tensorflow.
[28] 2018. *TensorFlow.* https://www.tensorflow.org/.
[29] 2018. *Tensorflow Models.* https://github.com/tensorflow/models.
[30] 2018. *Zalando Fashion MNIST.* https://github.com/cmasch/zalando-fashion-mnist.
[31] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007.*
[32] Constantino Carlos Reyes Aldasoro and Abhir Bhalerao. 2007. Volumetric texture segmentation by discriminant feature selection and multiresolution classification. *IEEE Transactions on Medical Imaging* 26, 1 (2007).
[33] Sanjeev Arora, Andrej Risteski, and Yi Zhang. 2017. Theoretical limitations of Encoder-Decoder GAN architectures. *arXiv:1711.02651* (2017).
[34] Sanjeev Arora and Yi Zhang. 2017. Do GANs actually learn the distribution? an empirical study. *arXiv:1706.08224* (2017).
[35] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.*
[36] Ivan Bocić and Tevfik Bultan. 2016. Finding access control bugs in web applications with CanCheck. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.*
[37] Mat Buckland and Mark Collins. 2002. *AI techniques for game programming.*
[38] Gabriel Cadamuro, Ran Gilad-Bachrach, and Xiaojin Zhu. 2016. Debugging machine learning models. In *ICML Workshop on Reliable Machine Learning in the Wild.*
[39] Aleksandar Chakarov, Aditya Nori, Sriram Rajamani, Shayak Sen, and Deepak Vijaykeerthy. 2016. Debugging machine learning tasks. *arXiv:1603.07292* (2016).
[40] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002).
[41] Fuxiang Chen and Sunghun Kim. 2015. Crowd Debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* ACM, New York, NY, USA, 320–332. https://doi.org/10.1145/2786805.2786819
[42] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in Neural Information Processing Systems.*
[43] Travers Ching, Daniel S Himmelstein, Brett K Beaulieu-Jones, Alexandr A Kalinin, Brian T Do, Gregory P Way, Enrico Ferrero, Paul-Michael Agapow, Michael Zietz, Michael M Hoffman, et al. 2018. Opportunities and obstacles for deep learning in biology and medicine. *bioRxiv* (2018).
[44] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.
[45] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2018. AutoAugment: Learning Augmentation Policies from Data. *arXiv preprint arXiv:1805.09501* (2018).
[46] George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2, 4 (1989).
[47] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 582–592.
[48] Quan Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Regression test selection for android applications. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on.*
[49] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-based dependency parsing with stack long short-term memory. *arXiv:1505.08075* (2015).
[50] Eran Eidinger, Roee Enbar, and Tal Hassner. 2014. Age and gender estimation of unfiltered faces. *IEEE Transactions on Information Forensics and Security* (2014).
[51] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.*
[52] Qiang Fu, Jian-Guang Lou, Qing-Wei Lin, Rui Ding, Dongmei Zhang, Zihao Ye, and Tao Xie. 2012. Performance issue diagnosis for online service systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on.* IEEE, 273–278.
[53] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering.* IEEE Press, 758–769.
[54] Robert Geirhos, David HJ Janssen, Heiko H Schütt, Jonas Rauber, Matthias Bethge, and Felix A Wichmann. 2017. Comparing deep neural networks against humans: object recognition when the signal gets weaker. *arXiv:1706.06969* (2017).
[55] Olivier Giroux and Martin P Robillard. 2006. Detecting increases in feature coupling using regression tests. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering.*
[56] Yoav Goldberg. 2016. A Primer on Neural Network Models for Natural Language Processing. *J. Artif. Intell. Res.(JAIR)* 57 (2016).
[57] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems.*
[58] Serkan Gunal and Rifat Edizkan. 2008. Subspace based feature selection for pattern recognition. *Information Sciences* 178, 19 (2008).
[59] Venkatesan Guruswami and Prasad Raghavendra. 2009. Hardness of learning halfspaces with noise. *SIAM J. Comput.* 39, 2 (2009).
[60] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. 2005. Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning. In *International Conference on Intelligent Computing.*
[61] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on.*
[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.*

[63] Mikael Henaff, Jason Weston, Arthur Szlam, Antoine Bordes, and Yann LeCun. 2016. Tracking the world state with recurrent entity networks. *arXiv:1612.03969* (2016).

[64] Felix Hill, Antoine Bordes, Sumit Chopra, and Jason Weston. 2015. The goldilocks principle: Reading children's books with explicit memory representations. *arXiv:1511.02301* (2015).

[65] Julia Hirschberg and Christopher D Manning. 2015. Advances in natural language processing. *Science* 349, 6245 (2015).

[66] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. 2007. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments.* Technical Report 07-49. University of Massachusetts, Amherst.

[67] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. 2017. Detecting Energy Bugs in Android Apps Using Static Analysis. In *International Conference on Formal Engineering Methods.*

[68] Yuan Jiang and Zhi-Hua Zhou. 2004. Editing training data for kNN classifiers with neural network ensemble. In *International symposium on neural networks.*

[69] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated behavioral regression testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on.*

[70] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering.*

[71] Taskin Kavzoglu. 1999. Determining Optimum Structure for Artificial Neural Networks. In *Proceedings of the 25th Annual Technical Conference and Exhibition of the Remote Sensing Society.*

[72] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on.*

[73] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering.* ACM.

[74] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society.

[75] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[76] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *International Conference on Machine Learning.*

[77] Sanjay Krishnan and Eugene Wu. 2017. PALM: Machine Learning Explanations For Iterative Debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics.*

[78] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

[79] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.*

[80] Todd Kulesza, Simone Stumpf, Margaret Burnett, Weng-Keen Wong, Yann Riche, Travis Moore, Ian Oberst, Amber Shinsel, and Kevin McIntosh. 2010. Explanatory debugging: Supporting end-user debugging of machine-learned programs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on.*

[81] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2012).

[82] Taek Lee, Jaechang Nam, Donggyun Han, Sunghun Kim, and Hoh Peter In. 2016. Developer micro interaction metrics for software defect prediction. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1015–1035.

[83] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press.

[84] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007).

[85] Zachary C Lipton. 2016. The mythos of model interpretability. *arXiv:1606.03490* (2016).

[86] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV).*

[87] Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. 2017. LAMP: data provenance for graph based machine learning algorithms through derivative computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.*

[88] Peter McCullagh. 1984. Generalized linear models. *European Journal of Operational Research* 16, 3 (1984).

[89] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv:1411.1784* (2014).

[90] Augustus Odena, Christopher Olah, and Jonathon Shlens. 2016. Conditional image synthesis with auxiliary classifier gans. *arXiv:1610.09585* (2016).

[91] Alessandro Orso, Hyunsook Do, Gregg Rothermel, Mary Jean Harrold, and David S Rosenblum. 2007. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability* 17, 2 (2007).

[92] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering.*

[93] C.C. Reyes-Aldasoro and A. Bhalerao. 2006. The Bhattacharyya space for feature selection and its application to texture segmentation. *Pattern Recognition* 39, 5 (2006), 812 – 826. https://doi.org/10.1016/j.patcog.2005.12.003

[94] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

[95] Abhik Roychoudhury and Satish Chandra. 2016. Formula-based software debugging. *Commun. ACM* 59, 7 (2016).

[96] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation.* Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.

[97] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[98] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1.

[99] Jacob Schrum and Risto Miikkulainen. 2015. Constructing Game Agents Through Simulated Evolution. *Encyclopedia of Computer Graphics and Games* (2015).

[100] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30.

[101] Shai Shalev-Shwartz and Shai Ben-David. 2014. Something May Go Wrong – Overfitting. In *Understanding machine learning: From theory to algorithms.* Chapter 2.2.

[102] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016).

[103] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 245–256.

[104] Amanda Swearngin, Myra B Cohen, Bonnie E John, and Rachel KE Bellamy. 2013. Human performance regression testing. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press.

[105] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning.. In *AAAI*, Vol. 4.

[106] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *arXiv preprint arXiv:1711.07163* (2017).

[107] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 63.

[108] Xin Xia, David Lo, Ying Ding, Jafar M Al-Kofahi, Tien N Nguyen, and Xinyu Wang. 2017. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering* 43, 3 (2017).

[109] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.

[110] Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.*

[111] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

[112] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering.*

[113] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *Software Engineering, 2003. Proceedings. 25th International Conference on.*

[114] Xuezhou Zhang, Xiaojin Zhu, and Stephen Wright. 2018. Training set debugging using trusted items. In *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI).*