Detect and Repair Errors for DNN-based Software

Yuchi Tian

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Detect and Repair Errors for DNN-based Software

Yuchi Tian

Nowadays, deep neural networks based software have been widely applied in many areas including traffic control, medical diagnosis, image recognition and malware detection, *etc.* However, the software engineering techniques, which are supposed to guarantee the functionality, safety as well as fairness, are not well studied. For example, serious real-world DNN based autonomous cars crashes have been reported. These crashes could have been avoided if these DNN based software were well tested. Traditional software testing, debugging and repairing techniques do not work on DNN based software because there is no control flow, data flow or AST(Abstract Syntax Tree) in DNNs. Proposing software engineering techniques targeted on DNN based software are imperative and in urgent need. In this thesis, we first introduced the development of SE(Software Engineering) for AI(Artificial Intelligence) area and how our works have influenced the advancement of this new area. Then we summarized related works and some important concepts in SE for AI area. Finally, we discussed four important works of ours.

Our first project DeepTest is one of the first few papers proposing systematic software testing techniques for DNN based software. We proposed neuron coverage guided test generation techniques for DNN based autonomous cars and leveraged domain specific metamorphic relation to generate oracle for new generated test cases to automatically test DNN based software. We applied DeepTest to testing three top performing self-driving car models in Udacity self-driving car challenge and identified thousands of erroneous behaviours that may lead to potential fatal

crash.

In DeepTest project, we found that the natural variation such as spatial transformations or rain/fog effects have led to problematic corner cases for DNN based self-driving cars. In the follow-up project DeepRobust, we studied per-point robustness of deep neural network under natural variation. We found that for a DNN model there are some specific weak points, which are more likely to cause errors under natural variation. We proposed a white-box approach and a black-box approach to identify these weak data points. We implemented and evaluated our approaches on 9 DNN based image classifiers and 3 DNN based self-driving car models. Our approaches can successfully detect weak points with good precision and recall for both DNN based image classifiers and self-driving cars.

Most of existing works in SE for AI area including our DeepTest and DeepRobust focus on instance-wise errors, which are single generated inputs that result in a DNN model's erroneous behaviors. Besides instance-wise errors, we found another type of errors, group-level errors, which reflect a DNN model's weak performance on differentiating among certain classes or inconsistent performance across classes. This type of errors is very concerning since it has been found to related to many real-world notorious errors without malicious attackers. In our third project DeepInspect, we first introduced the group-level errors which DNN based software also suffer from and categorized them into *confusion errors* and *bias errors* based on real-world reports. Then we proposed neuron coverage based distance metric to detect group-level errors for DNN based software without the need of data labels. We applied DeepInspect to testing eight pretrained DNN models trained in six popular image classification datasets, including three adversarial trained models. We showed that DeepInspect can successfully detect group-level violations for both single-label and multi-label classification models with high precision.

As a follow-up and more challenging research project, we proposed five WR(weighted regularization) techniques to repair group-level errors for DNN based software. These five different weighted regularization techniques function at different stages of retraining or inference of DNNs including input phase, layer phase, loss phase and output phase. We compared and

evaluated these five different WR techniques in both single-label and multi-label DNN-based classifications including five combinations of four DNN architectures on four datasets. We showed that WR can effectively fix confusion and bias errors and these methods all have their pros, cons and applicable scenario.

All our four projects discussed in this thesis have solved important problems in ensuring the functionality, safety as well as fairness for DNN based software and had significant influence in the advancement of SE for AI area.

Table of Contents

Acknowledgements

First and foremost, I would like to express my deep gratitude to my advisor Professor Baishakhi Ray for her continuous support and invaluable guidance throughout my Ph.D. studies. She has been motivating, inspiring and advising me in my Ph.D. researches. Without her guidance and help, this dissertation would not have been possible. This thesis includes a collection of collaborative works. I especially thank Professor Suman Jana and Kexin Pei for their collaboration in the first project DeepTest. I enjoy the collaboration with Ziyuan Zhong in the second project DeepRobust and thank him for the collaboration in the third and fourth project. I also thank Professor Vicente Ordóñez and Professor Gail Kaiser for their insightful discussions and collaboration in the third and fourth project.

Secondly, I would like to thank colleagues including Ziyuan Zhong, Yangruibo Ding, Chengzhi Mao, Saikat Chakraborty, Renqin Cai, Weilin Xu and Aihua Chen for their help through the journey during my Ph.D. studies. Besides, our works are partly supported by NSF grants, ONR grants and Google Faculty Fellowship.

Finally, I am grateful to my wife for her understanding, thoughtfulness and patience, my mother for her selfless support in every stage of my life and my father for his encouragement and being a great role model.

Chapter 1: Introduction

In recent years, deep neural networks have been developed and adopted in many areas including computer vision, natural language processing, program synthesis, recommending systems, *etc.* and even in some safety-critical areas, such as self-driving cars, medical diagnosis, malware detection, *etc.* However, these DNN based software have not been adequately tested and thus the functionality, safety as well as fairness are not guaranteed. This is because existing software testing techniques for DNN based software are very limited and most of deep learning researchers and developers only focus on improving the accuracy in test data[1, 2, 3, 4] and neglect other software quality measurement. Most of traditional software testing techniques do not work for DNN based software. The DNN based software, unlike traditional software where program logic is manually written by software developers, learn its logic automatically from the training data. The learned weights as well as model structures (neurons and layers) are much harder to interpret and thus challenging for debugging. The code coverage such as statement coverage, branch coverage, MC/DC coverage[5, 6, 7], *etc.*, which are used for measuring how well traditional software are tested, do not work for DNNs at all. The test generation techniques, such as MHS(metaheuristic search)[8] based test generation[9, 10, 11, 12] and the symbolic execution based test generation[13, 14, 15, 16, 17, 18], *etc.* do not work for DNNs because the complex input type such as image or video and there is no control flow or data flow in DNNs. There are some software testing techniques for machine learning based software, which can be adapted to DNN based software, such as applying metamorphic testing to machine learning applications without test oracles[19, 20, 21, 22]. However, these software testing techniques do not leverage information specifically in DNNs such as neurons, gradients, weights, loss functions and *etc.* and thus less effective. Similarly, existing automatic program repairing techniques are also only targeted to traditional software.[23]. These techniques include random or guided mutation of AST(Abstract Syntax Tree)[24, 25, 26, 27, 28,

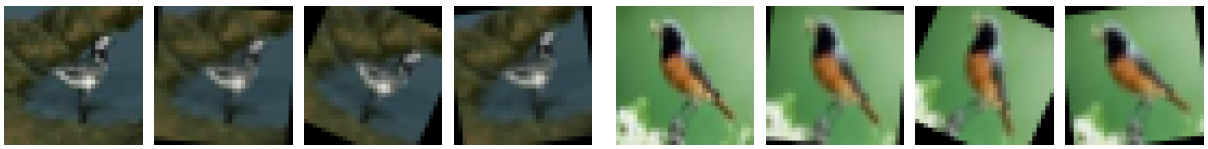
29, 30, 31], static program analysis or symbolic execution/concrete execution[32, 33, 34, 35, 36, 37] and most recently, language models training and program synthesis[38, 39]. All these techniques for repairing traditional programs such as C, C++ or Java, cannot work on DNN based software because there is no control flow, data flow or AST in deep neural networks.

However, just like traditional software, DNN-based software suffer from unexpected corner cases that can lead to dangerous consequences like a fatal collision in self-driving cars. Several such real-world cases have already been reported (see Table 1.1). For example, the fatal Tesla crash resulted from a failure to detect a white truck against the bright sky. Such severe crashes could be avoided if the DNN based autonomous cars are well tested. The existing possible approaches to detect such bugs depend heavily on manual collection of labeled test data and techniques to measure how well the DNN based software is tested are not available. The systematic testing tools for DNN based software are far away from being ready, not mentioning the repairing techniques for repairing DNN based software. Therefore, proposing software engineering techniques for DNN based software are imperative and in urgent need.

Table 1.1: **Examples of real-world accidents involving autonomous cars**

	Reported Date	**Cause**	**Outcome**	**Comments**
Hyundai Competition [40]	December, 2014	Rain fall	Crashed while testing	"The sensors failed to pick up street signs, lane markings, and even pedestrians due to the angle of the car shifting in rain and the direction of the sun" [40]
Tesla autopilot mode [41]	July, 2016	Image contrast	Killed the driver	"The camera failed to recognize the white truck against a bright sky" [42]
Google self-driving car [43]	February, 2016	Failed to estimate speed	Hit a bus while shifting lane	"The car assumed that the bus would yield when it attempted to merge back into traffic" [43]

Initially, DNNs are mostly developed by machine learning researchers, who are interested in improving the performance with respect to test accuracy. In this thesis, we argue the necessity of more software engineering measurement and software engineering techniques that can work for DNN based software. It is significantly important to design, implement and evaluate new systematic testing tools that can work for DNN based software to assure the functionality, safety as well as fairness. One of our work DeepTest to be discussed in Chapter 3 is one of the first few paper applying software engineering techniques on DNN based software. We proposed neuron coverage guided image synthesis techniques to generate test cases for DNN based software and at the same



(a) 0°, bird (b) +6°, airplane (c) +24°, cat (d) -9°, dog (e) 0°, bird (f) +6°, bird (g) +24°, bird (h) -9°, bird

Figure 1.1: **(a)-(d) A well-trained ResNet model [47] misclassifies the rotated variations of a bird image into three different classes though the original un-rotated image is classified correctly. (e)-(h) The same model successfully classifies all the rotated variants of another bird image from the same test set. The sub-captions consist of rotation degrees and the predicted classes.**

time maximize the coverage of new neurons. Different from DeepXplore[44], which requires multiple implementations required by differential testing, ours is able to test single implementation by leveraging metamorphic testing to generate labels for those new generated test cases as the test oracle to automatically test DNN based software. We apply DeepTest to testing top-rank DNN based self-driving car models in Udacity self-driving car challenge and our tool discovers thousands of erroneous behaviours that may lead to potential fatal crash. We also retrain these self-driving car models with our new generated test inputs and show that those bugs can be fixed after retraining.

Deep neural networks are extremely vulnerable to natural variation such as spatial transformations or rain/fog effects. The natural variants are especially concerning as they can occur naturally in the field without any active adversary and may lead to serious consequences [45, 46]. In our second project DeepRobust, we tried to understand per-point robustness of deep neural network under natural variation. We found that not all the inputs under natural variation will result in erroneous outputs for a DNN. There are specific "weak" data points, which are more likely to fail a deep neural network than other data points. For example, consider Figure 1.1: although the original bird image (a) is predicted correctly by a DNN, its rotated variations in images (b)-(d) are mispredicted to three different classes. This makes the original image (a) very weak as far as robustness is concerned. In contrast, the bird image (e) and all its rotated versions (generated by the same degrees of rotation) in Figure 4.1:(f)-(h) are correctly classified. Thus, the original image (e) is quite robust. It is important to distinguish between such robust vs. non-robust images, as the

non-robust ones can induce errors with slight natural variations.

Existing works in DNN robustness mainly focuses on evaluating the overall robustness of DNNs across all the test data [47, 48, 49]. This is analogous to estimating how buggy a software is without actually localizing the bugs in traditional software. We argued that DNN is a combination of data and architecture. DeepRobust(Chapter 4) focuses on localizing the non-robust points in the input space that pose significant threats to a DNN model's robustness. Different from traditional software where bug localization is performed in program space, we identify the non-robust inputs in the data space. To address this problem, we proposed a white-box approach(DEEPROBUST-W) and a black-box approach(DEEPROBUST-B) to identify these weak data points for DNN based software. We applied our approaches to testing 9 DNN based image classifiers and 3 DNN based self-driving car models. Our results show that DEEPROBUST-W and DEEPROBUST-B are able to achieve an F1 score of up to 91.4% and 99.1%, respectively in testing DNN based image classifiers. DEEPROBUST-W is effective in identifying weak data points with F1 score up to 78.9% in testing DNN based self-driving car models.

More and more papers have been published in SE for AI area, as the software engineering techniques are urgently demanded for machine learning based software. Similar to code coverage in traditional software, different coverage for DNNs such as neuron coverage[44], k-multisection coverage and boundary coverage[50], SS(sign-sign coverage) coverage[51], *etc.* have been proposed and evaluated for DNNs. Similar to test generations in traditional software, norm based perturbation[52, 44] , natural transformations[53, 47] based and GAN(Generative Adversarial Networks)[46] based test generation have been proposed to generate corner cases input for testing DNNs. Metamorphic testing[53, 46], fuzzing testing[52, 54, 55], mutation testing[56], differential testing[44] and concolic testing[57, 58], *etc.* have been proposed for DNN based software. Abstract interpreter[59] has been proposed to verify DNN based software. Image transformations[47, 60] or data augmentation techniques[61, 55] are proposed to evaluate the robustness of DNN based software.

Besides of the new testing techniques for DNN based software, the repairing techniques for

DNN based software have been studied too[62]. Different data augmentation and fine-tuning techniques have been proposed for repairing DNN models in improving overall accuracy[63, 64, 65]. There are also works in improving robustness of DNN models against adversarial instances[55, 60, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75].

However, existing studies including our DeepTest and DeepRobust in detecting and repairing errors only focus on instance-wise errors, while group-level errors are mostly ignored. An instance-wise error happens when a DNN model outputs inconsistent prediction given a specific input[44, 45, 76, 72]. These inputs include adversarial examples from norm-bounded perturbation[72], realistic transformation[47] of an existing input, or physical attack[77], *etc.*. To repair these instance-wise errors, techniques such as adversarial training, data augmentation, *etc.* are widely leveraged [72, 47]. In contrast, group-level error is about the DNN model's weak performance on differentiating among certain classes or has inconsistent performance across classes[78]. This type of bugs is very concerning since it has been found to relate to many real-world notorious errors without malicious attackers[78] as shown in Table 1.2. For example, Google faced backlash in 2015 due to a notorious error in its photo-tagging app, which tagged pictures of dark-skinned people as "gorillas" [79]. Analogous to traditional software bugs, the Software Engineering (SE) literature denotes these classification errors as *model bugs* [65], which can arise due to either imperfect model structure or inadequate training data.

Table 1.2: **Examples of real-world bugs reported in neural image classifiers**

Bug Type	Name	Report Date	Outcome
Confusion	Gorilla Tag [79]	Jul 1, 2015	Black people were tagged as gorillas by Google photo app.
	Elephant is detected in a room [80]	Aug 9, 2018	Image Transplantation (replacing a sub-region of an image by another image containing a trained object) leads to mis-classification.
	Google Photo [81]	Dec 10, 2018	Google Photo confuses skier and mountain.
Bias	Nikon Camera [82]	Jan 22, 2010	Camera shows bias toward Caucasian faces when detecting people's blinks.
	Men Like Shopping [83]	July 29, 2017	Multi-label object classification models show bias towards women on activities like shopping, cooking, washing, *etc.*
	Gender Shades[84]	2018	Open-source face recognition services provided by IBM, Microsoft, and Face++ have higher error rates on darker-skin females for gender classification.

In one of our works, DeepInspect, to be discussed in Chapter 5, we investigate some public reports describing the class-level violations listed in Table 1.2, and categorize them into two group-level errors: (i) **Confusion Errors**: The model cannot differentiate one class from another. For

example, Google Photos confuses skier and mountain [81]. (ii) **Bias Errors**: The model shows disparate outcomes between two related groups. For example, Zhao *et al.* in their paper "Men also like shopping" [83], find classification bias in favor of women on activities like shopping, cooking, washing, *etc.* We further notice that in the case of *confusion errors*, the classification error-rate between the objects of two classes, say, *cat* and *dog*, can be significantly higher than the overall classification error rate of the model trained in CIFAR-10 dataset. In the bias scenario reported by Zhao *et al.*, a DNN model, while classifying the gender of a person, may mistakenly associate gender with a given specific activity, object or environment such as *shopping*, *sports* or *kitchen*. Different from instance-wise errors, this is a class level property affecting all the *shopping*, *sports* or *kitchen* images with men or women. Any violation of such a property by definition affects the whole class although not necessarily every image in that class, *e.g.,* a man is more prone to be predicted as a woman when he is shopping, in the kitchen or hold a baby, even though some individual images of a man may still be predicted correctly.

In DeepInspect, we also propose a novel neuron-coverage metric to automatically detect group-level violations (confusion and bias errors) in DNN-based models for image classification. Our tool DeepInspect found many errors in widely-used DNN models with precision up to 100% (avg. 72.6%) for confusion errors and up to 84.3% (avg. 66.8%) for bias errors. DeepInspect shows that these group-level errors occur in all popular models trained in popular classification dataset including single-label image classification where each image is labeled with one object and multi-label image classification where each image is associated with a set of labels. The following lists some examples of group-level errors DeepInspect identified in DNN models trained in popular image classification datasets. Figure 1.2, 1.3, 1.4 and 1.5 presents examples of confusion errors respectively from COCO, ImageNet, CIFAR-10 and CIFAR-100. Figure 1.6, 1.7 presents examples of bias errors respectively from COCO gender and ImSitu dataset.

A much harder and reasonable follow-up problem to solve is how to repair these errors. One of the root causes is that certain classes are harder to be differentiated from each other. For example, in CIFAR-10, *dog* and *cat* tend to confuse even a state-of-the-art DNN model since they share

(a) laptop -> laptop,mouse (b) bus -> bus,traffic lights (c) skis -> snowboard

Figure 1.2: **Examples of confusion errors found in COCO dataset**

(a) cello -> violin (b) violin -> cello (c) wok -> pan

Figure 1.3: **Examples of confusion errors found in ImageNet dataset**

(a) bird -> deer (b) bird -> deer (c) cat -> dog (d) cat -> dog

Figure 1.4: **Examples of confusion errors found in CIFAR10**

many common semantic features. For multi-label classification, one of the root causes is that two classes may appear together frequently. For example, in COCO dataset, mouse and laptop appear in the same image frequently, which make it hard for DNN model to distinguish between them. The confusing pair of classes tend to be very close with each other in the representation space and the decision boundary between them might not be "fine-grained" enough for correct classification on these confusing pairs. We denote the error-inducing classes as target classes. To fix the errors

(a) boy -> girl (b) boy -> girl (c) oak -> maple (d) oak -> maple

Figure 1.5: **Examples of confusion errors found in CIFAR100**

(a) given surfboard, woman -> man (b) given frisbee, woman -> man

Figure 1.6: **Examples of bias errors found in COCO gender dataset**

(a) given outside, woman -> man (b) given room, man -> woman

Figure 1.7: **Examples of bias errors found in ImSitu dataset**

of the target classes, the model needs to take more effort to learn from them.

In the last work(Chapter 6) to be discussed in this thesis, We propose a generic method called weighted regularization (WR) to repair group level errors in DNN based software. WR consists of

five weighted regularization methods including weighted augmentation (w-aug), weighted batch normalization (w-bn), weighted output smoothing (w-os), weighted loss (w-loss), and weighted distance-based regularization (w-dbr). These methods function at different stage of a given DNN's training or inference as shown in Figure 1.8. Specifically, if retraining is allowed and training data are accessible, w-aug assigns more weights to the target classes during the retraining; w-bn re-scales the distribution of the activation values induced by the input at every batchnorm layer to shift the decision boundary toward target classes(assuming the model has batchnorm layers); w-loss modifies the loss function by assigning more weights to the mis-classification occurring between target classes while w-dbr updates the loss function by regularizing the class centroids in the representation space. Such regularization strategies enable the model to emphasize more on the instances of the target classes and thus reduce the errors between the target classes. If fine-tuning is not possible or training data are not accessible, w-os multiplies the model's prediction on target classes by a small user-specified constant to make the model predict less the target class such that those unsure data points located in decision boundary(mis-classification between target classes) can be avoided.

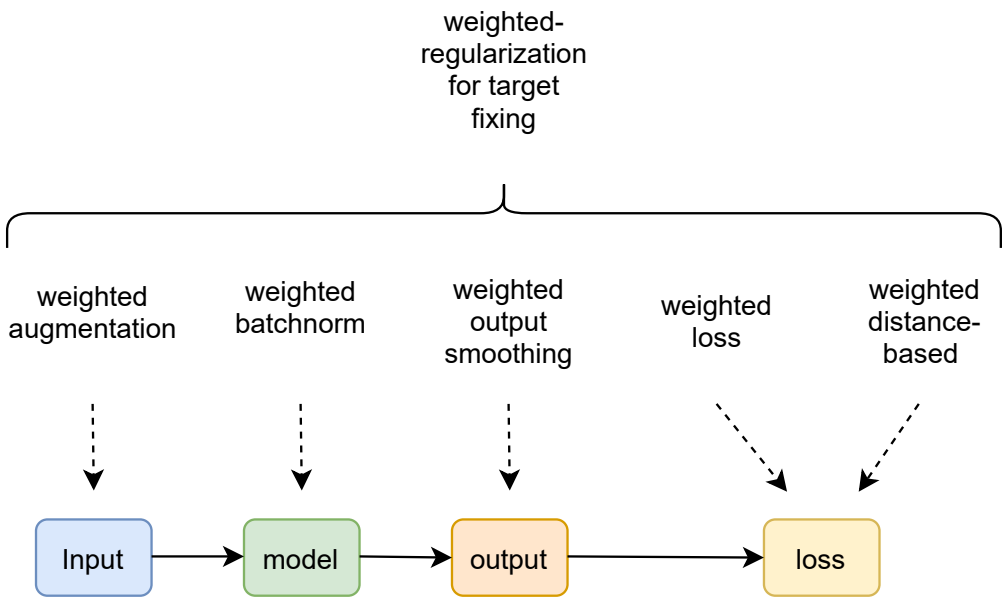


Figure 1.8: **Overview of Weighted Regularization for Target Fixing**

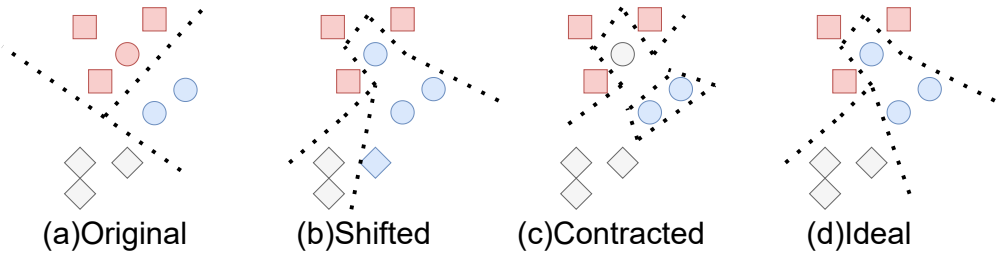


Figure 1.9: **Illustration of different potential decision boundary before and after applying WR.**

Figure 1.9 illustrates how these different approaches function using an example consists of three classes (square, circle and diamond). The colors represent the model's prediction while the dashed lines denote the model's decision boundary. Figure 1.9(a) shows that the original model tends to confuse between square and circle. The ideal fix of decision boundary is shown in Figure 1.9(d). w-os solves the confusion error by contracting the decision boundary of the target classes as illustrated in Figure 1.9(c). w-aug, w-loss, and w-dbr try to reduce confusion by shifting the decision boundary. They may be able to achieve Figure 1.9(d) but it may also be possible to sacrifice the decision boundary for other classes and get the decision boundary in Figure 1.9(b) instead. w-bn comes in between: on the one hand, it tends to contract the decision boundary as w-os; on the other hand, it tends to shift the decision boundary through fine-tuning. We evaluate different regularization methods and compare their effectiveness on four widely-used datasets and architectures and show that WR can effectively fix confusion and bias errors and these approaches all have their pros, cons and applicable scenario. The detail results will be discussed in Chapter 6.

In summary, this thesis incorporates four major projects in SE for AI area.

- Chapter 3: DeepTest(Automated Testing of DNN based Autonomous Cars), published in ICSE' 18[45]
- Chapter 4: DeepRobust(Understanding Local Robustness of Deep Neural Networks under Natural Variations), published in FASE' 21[76]
- Chapter 5: DeepInspect(Automated Testing Group Level Errors for DNN based Image Classifiers), published in ICSE' 20[78]

- Chapter 6: WR (Repair Group Level Errors for DNN based Image Classifiers Using Weighted Regularization), one short paper published in FSE' 20[85], another 12-page paper in submission in ASE' 21.

1.1 Problem Statement

Our studies focus on proposing new software engineering techniques to ensure the functionality, safety as well as fairness for DNN based software.

In recent years, more and more papers have been published in this SE for AI area from coverage metric for DNNs to automatic testing, repairing and verification for DNN based software. Here we discuss existing works' limitation, specify the problems our projects try to solve and summarize our contributions.

- Before our DeepTest project, there is very few paper in SE for AI area. Most of works were trying to improve DNNs' generalization with respect to natural accuracy in deep learning studies[1, 2, 3, 4] or robust accuracy in adversarial examples studies[61, 86]. Pei *et al.* in DeepXplore proposed neuron coverage to measure how well a DNN based software is tested, leveraged gradient based perturbation to generate test inputs to test image classifiers[44]. Our project DeepTest(Chapter 3) presents an neuron coverage based grey-box approach to guide our generation of test inputs. In contrast to gradient based perturbation by DeepXplore we are using nature image transformations including rain or fog weather to simulate the camera shake, different weather conditions, *etc.* Our generated test inputs are more natural since the image transformations can very well simulate the real-world driving conditions.

- Pei *et al.* in DeepXplore applied differential testing from software engineering in testing different DNN models. However, differential testing requires multiple DNN models. We proposed domain specific metamorphic relation to generate oracles for all the synthesized images, which enables the testing of single DNN model in DeepTest(Chapter 3). DeepTest applies metamorphic testing from software engineering to testing the DNN driven autonomous

cars as one of the first few paper in SE for AI area.

- As a follow-up project of DeepTest(Chapter 3), we conducted an empirical study on per-point robustness under natural variation to dig further into DNNs' vulnerability against natural transformations in DeepRobust(Chapter 4). Existing works in DNN robustness, focuses on evaluating the overall robustness of DNNs across all the test data [47, 48, 49]. This is analogous to estimating how buggy a software is without actually localizing the bugs in traditional software. DeepRobust(Chapter 4) tries to localize the non-robust points in the input space that pose significant threats to a DNN model's robustness. However, unlike traditional software where bug localization is performed in program space, we identify the non-robust inputs in the data space. We proposed white-box and black-box solutions in identifying non-robust points for both DNN based image classifiers and DNN based self-driving car models.

- In recent years, more and more papers have been published in DNN testing, repairing as well as verification. However, all of these works focus on instance-wise errors, while group-level errors are completely ignored. An instance-wise error is a classification mistake a DNN model makes, given a specific input. This input could be an original test case or an adversarial test case from norm-bounded perturbation[72], natural transformation[47], or physical attack[77] *etc.* In contrast, group-wise error is about the DNN model's weak performance on differentiating among certain classes or has inconsistent performance across classes. Our paper DeepInspect(Chapter 5) categorizes group-level errors into confusion errors and bias errors and proposes neuron coverage based metric to detect confusion errors and bias errors in DNN based single-label and multi-label classifiers.

- A reasonable follow-up and more challenging problem is to repair group level errors for DNN based software. Our last project(Chapter 6) proposed five different weighted regularization techniques to address this problem. These five different techniques are leveraged in different stages of DNN retraining or inference including input phase, layer phase, output phase and loss phase. We compare and evaluate these five different WR techniques in both

single-label and multi-label DNN-based classifications including five combinations of four DNN architectures for four datasets. The results show that WR can effectively fix confusion and bias errors and these methods all have their pros, cons and applicable scenario.

1.2 Thesis Overview

Following this chapter, we include a detail literature review in the SE for AI areas where our works focus on and necessary background knowledge in computer vision, machine learning and software engineering to prepare readers for following chapters' discussion. In Chapter 3, we presented our project DeepTest, which, to our knowledge, is the first systematic testing tool for DNN based autonomous cars. We will discuss our neuron coverage guided inputs synthesize algorithm and how we apply metamorphic testing in DNNs as well as the implementation and evaluation of our tool DeepTest on top performing models in Udacity self-driving car challenge. In Chapter 4, we presented our project DeepRobust in understanding the local robustness of deep neural networks under natural variation. We showed our finding that only specific weak points would result in errors under natural variation. We proposed a white-box approach and black-box approach to identify these weak points for DNN based image classifiers and DNN based self-driving cars. In Chapter 5, we presented our project DeepInspect, which introduces the concepts of group-level errors in DNNs and categorizes them into confusion errors and bias errors. We will discuss the difference between group-level errors and instance-wise errors. We will also discuss the methodology and evaluation of DeepInspect, a systematic tool to identify group-level errors for DNN based image classifiers. In Chapter 6, we presented our project "Repairing group-level errors using weighted regularization". In this chapter, we discuss the five weighted regularization approaches, which works on different stages of retraining and inference and thus can be applied in different scenarios to effectively repair group-level errors. We conclude this thesis with a summary of contribution, discussion of existing works' limitation, potential future works.

Chapter 2: Background and Related Work

Our works discussed in this thesis explored and proposed new software engineering techniques for DNN based software to ensure the functionality, security as well as fairness. First in Section 2.1, we briefly summarized traditional software engineering techniques, explained the difference between traditional software and DNN based software and show why traditional software techniques cannot work on DNN based software. In Section 2.2, we introduced the popular DNN architectures and two DNN based software, DNN based autonomous cars and DNN based image classifiers. DNN based autonomous car is one of most popular safety critical DNN applications in recent years and its safety issues are very concerning. We introduced DNN based autonomous cars to prepare readers for our first project in proposing systematic testing techniques for DNN based autonomous cars. Image classification is one of the most popular DNN applications. We introduced two types of image classification tasks, single-label/multi-class classification and multi-label classification to prepare readers for our third and fourth project in detecting and repairing errors in DNN based image classifiers. Both DNN based autonomous cars and DNN based image classifiers are useful for understanding the second project because the second project proposed approaches in localizing non-robust points for both DNN based applications. Then in Section 2.3, we discussed related works in DNN testing and repairing techniques. There are generally two categories of errors in DNN testing and repairing, instance-wise errors and group level errors. We showed that most of existing works in SE for AI focus on instance-wise errors and motivated readers about the importance of group level errors to prepare readers for our third and fourth project in proposing techniques in testing DNN based software with respect to group level errors. Lastly in Section 2.4, we discussed the related works in DNN bias/fairness to prepare readers for our third and fourth project in detecting bias errors and repairing bias triples for DNN based software.

2.1 Software Testing and Repairing

2.1.1 Test Generation and Test Criteria

Software testing for traditional software usually involves test generation, amplification and leveraging different test criteria to measure the quality of a test set or how well the software has been tested. There is a large body of work on test case generation and amplification techniques for traditional software that automatically generate test cases from some seed inputs and increase code coverage. Instead of summarizing them individually here, we refer the interested readers to the surveys by Anand et al. [87], McMinn et al. [9], and Pasareanu et al. [88]. There are also well-established test criteria for traditional software including statement coverage, branch coverage, MC/DC coverage[5, 6, 7], *etc.*. However, these test generation and amplification techniques as well as test criteria for traditional software cannot work for DNN based software because there is no control flow or data flow in DNNs and the input type for DNN based software is usually much more complex such as image or video. Inspired from test amplification for traditional software, one of the contributions in our work DeepTest(Chapter 3) is to automatically generate test inputs for DNN based autonomous cars and at the same time maximize the neuron coverage. In our second project DeepInspect(Chapter 5), we propose neuron coverage based metric to test DNN based image classifiers for group-level errors/bugs. The approaches we propose can be generalized for other DNN based software as well, although the evaluation is targeted to a specific type of software application.

2.1.2 Software Repairing.

For the automatic program repairing techniques that target traditional software, we refer interested readers to this review[23]. In summary, automatic program repairing techniques include random or guided mutation of AST(Abstract Syntax Tree)[24, 25, 26, 27, 28, 29, 30, 31], static program analysis or symbolic execution/concrete execution[32, 33, 34, 35, 36, 37] and most re-

cently, language models training and program synthesis[38, 39]. All these techniques for repairing traditional programs such as C, C++ or Java, cannot work on DNN based software because there is no control flow, data flow or AST in deep neural networks. In our third major project(Chapter 6), we propose weighted regularization techniques, which can automatically repair DNN based software for group-level errors.

2.2 DNN based Software

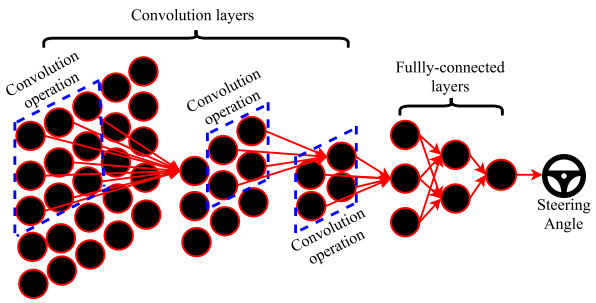
2.2.1 DNN Architectures

Most popular DNNs used in DNN based software can be categorized into two types: (1) Convolutional Neural Network (CNN), and (2) Recurrent neural network (RNN). We provide a brief description of each architecture below and refer the interested readers to [89] for more detailed descriptions.

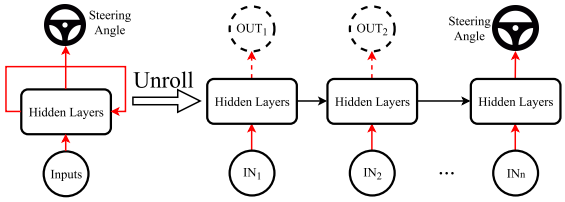
CNN architecture. The most significant signature to distinguish a CNN from a fully connected neural network is the *convolution layers*. The neurons in a convolution layer are connected only to a subset of the neurons in the next layer and multiple connections share the same weight. The sets of connections sharing the same weights are essentially a convolution kernel [90] that applies the same convolution operation on the outputs of a set of neurons in the previous layer. Figure 2.1a illustrates a simplified CNN based self-driving car architecture with three convolution layers, which is similar to the models used in practice [91].

CNNs have much fewer trainable parameters than fully connected neural networks by allowing sharing of weights among multiple connections and thus the training process of CNNs is much faster. CNNs perform especially well for image or video input as their architecture resembles the human visual system which extracts a layer-wise representation of visual input [90, 92].

RNN architecture. In RNNs[93], the output of each layer is not only fed to the following layer but also flow back to the previous layer. Such arrangement allows the prediction output for previous inputs (*e.g.,*, previous frames in a video sequence or previous words in a sentence) to be also considered in predicting current input.



(a) A simplified CNN architecture



(b) A simplified RNN architecture

Figure 2.1: (Upper row) A simplified CNN architecture with a convolution kernel shown on the top-left part of the input image. (Lower row) A simplified RNN architecture with loops in its hidden layers.

Figure 2.1b illustrates a simplified version of the RNN based self-driving car architecture.

Similar to other DNNs, RNNs also leverage gradient descent to update weights during back propagation for training. However, it is well known that the gradient, when propagated through multiple loops in an RNNs, may vanish to zero or explode to an extremely large value [94] and therefore may lead to an inaccurate model. *Long short-term memory* (LSTM) [95], a popular subgroup of RNNs, is designed to solve this vanishing/exploding gradient problem. We encourage interested readers to refer to [95] for more details.

2.2.2 DNN based Autonomous Cars

Our first major project DeepTest(Chapter 3) proposed techniques for automatically testing DNN based autonomous cars. Here we briefly introduce DNN based autonomous cars. The key component of an autonomous vehicle is the perception module controlled by the underlying Deep Neural Network (DNN) [96, 97]. The DNN takes input from different sensors like camera, light sensors and LiDAR sensors, and so on, which can measure the environment, and outputs the steering angle, braking, acceleration, *etc.* to maneuver the car safely in road as shown in Figure 2.2.

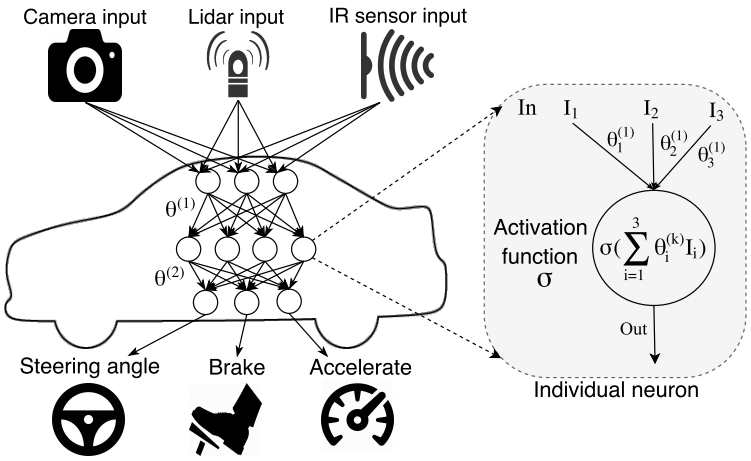


Figure 2.2: A simple autonomous car DNN that takes inputs from camera, light sensors and LiDAR sensors, and *etc.*, and outputs steering angle, braking, and acceleration control. The DNN models the function $\sigma(\boldsymbol{\theta}^{(2)} \cdot \sigma(\boldsymbol{\theta}^{(1)} \cdot \boldsymbol{x}))$ where $\boldsymbol{\theta}$s represent the weights of the edges and σ is the activation function. The details of the computations performed inside a single neuron are shown on the right.

DNNs(including CNNs and RNNs) are composed of multiple layers stacked together to extract different level of representations of the input [98]. Each layer of the DNN increasingly abstracts the input, *e.g.,*, from raw pixels to semantic concepts. For example, the first few layers of an autonomous car DNN extract low-level features such as edges and directions, while the deeper layers identify objects like stop signs and other cars, and the final layer outputs the steering decision (*e.g.,*, turning left or right by some degree).

Each layer of a DNN consists of multiple computing units called *neurons*. The neurons between adjacent layers are connected through edges, which has a corresponding weight ($\boldsymbol{\theta}$s as shown in Figure 2.2). Each neuron applies a nonlinear *activation function* on its inputs and sends the output to the subsequent neurons as shown in Figure 2.2. Popular activation functions include ReLU (Rectified Linear Unit) [99], Sigmoid [100], etc. The weights of a DNN is learned by backpropagation during the training process. Gradient descent is usually adapted in backpropagation [101]. After training, an inference of DNN with learned weights can be used for predictions. For example, an autonomous car DNN can predict the steering angle based on input images.

2.2.3 DNN based Image Classifier

Our second major project(Chapter 5) and third major project(Chapter 6) focus on testing and repairing DNN based image classifiers respectively. Image classification is one of the most popular applications of deep neural networks. There are generally two type of image classification tasks.

(i) **Single-label Classification.** In single-label classification problem, each datum is associated with a single label l from a set of disjoint labels L where $|L| > 1$. If $|L| = 2$, the classification problem is called a binary classification problem; if $|L| > 2$, it is a multi-class classification problem [102]. Among some popular image classification datasets, MNIST, CIFAR-10/CIFAR-100 [103] and ImageNet [104] are all single-label, where each image can be categorized into only one class or outside that class.

(ii) **Multi-label Classification.** In a multi-label classification problem, each datum is associated with a set of labels Y where $Y \subseteq L$. COCO[105] and imSitu[106] are popular datasets for multi-label classification. For example, an image from the COCO dataset can be labeled as *bus, person, traffic light*. A multi-label classification model is supposed to predict all of *bus, person, traffic light* from a single image that shows all of these kinds of objects.

Given any single- or multi-label classification task, DNN based classifier is supposed to learn the decision boundary between different classes—all members of a class, say C_i, should be categorized identically irrespective of their individual features, and members of another class, say C_j, should not be categorized to C_i [107]. The DNN represents the input image in an embedded space with the feature vector at a certain intermediate layer and uses the subsequent layers as a classifier to classify these representations. The *class separation* between two classes estimates how well the DNN has learned to separate each class from the other. If the distance in an embedded feature space between two classes is too small compared to other classes, or lower than some pre-defined threshold, the DNN can hardly separate them from each other.

2.3 DNN Testing and Repairing

2.3.1 Against Instance-wise Errors

Most of existing works in DNN testing focus on identifying instance-wise errors. An instance-wise error happens when a DNN model outputs inconsistent prediction across different semantic-preserving transformations of a given input [44, 45, 76, 72]. Testing of DNN based software for instance-wise errors usually involves generating test cases using norm-bounded perturbation[72], natural transformation[47], or physical attack[77], *etc.*. Similar to coverage metric in traditional software, coverage metrics such as neuron coverage[44], k-multisection coverage[50], neuron boundary coverage[50] and sign-sign coverage[108] have been proposed for DNN based software to evaluate how well these software are tested. Different white-box and black-box testing approaches based on these new metrics have been proposed to identify instance-wise errors for DNN based software [44, 53, 46, 57, 52].

Adversarial deep learning studies also target instance wise errors. DNNs are known to be vulnerable to well-crafted inputs called adversarial examples, where the discrepancies are imperceptible to a human but can easily fool deep neural network models [109, 110, 111, 112, 77, 113, 114, 115, 116, 117, 118, 119, 120, 121]. These adversarial examples can be used to test DNN based software. However, they are manually crafted and may not exist in reality. Our work DeepTest(Chapter 3) leveraged image transformations to simulate camera shake, weather condition, *etc.*. Our generated test cases are more realistic and at the same time maximize neuron coverage. Our follow-up project DeepRobust(chapter 4) studied per-point robustness of neural network under natural transformations and proposed both white-box and black-box solutions to identify non-robust data points for DNN based image classifiers and self-driving cars.

There are also studies in repairing DNN based software against instance-wise errors or defending against adversarial attacks by using adversarial training or data augmentation [122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 75, 72, 47]. Our work DeepTest(Chapter 3) also shows that retraining with our generated test cases can repair the DNN based software.

2.3.2 Against Group-level Errors

Group-level errors are about the DNN model's weak performance on differentiating among certain classes or has inconsistent performance across classes[78]. For example, Google faced backlash in 2015 due to a notorious error in its photo-tagging app, which tagged pictures of dark-skinned people as "gorillas" [79]. Our third project(Chapter 5) investigates some public reports describing the class-level violations listed in Table 1.2, and categorize them into two group-level errors: (i) **Confusion Errors**: The model cannot differentiate one class from another. For example, Google Photos confuses skier and mountain [81]. (ii) **Bias Errors**: The model shows disparate outcomes between two related groups. For example, a DNN model should not have different error rates while classifying the gender of a person, given a specific activity, object or environment such as *shopping*, *sports* or *kitchen*. Unlike individual image properties, this is a class property affecting all the *shopping*, *sports* or *kitchen* images with men or women. Any violation of such a property by definition affects the whole class although not necessarily every image in that class. Our third project(Chapter 5) is the first paper to propose techniques in testing DNN based software against group level errors. Our fourth project(Chapter 6) proposed weighted regularization approaches to repair DNN based software against target confusion errors or bias errors.

2.4 Bias/Fairness Related Work

Evaluating Models' Bias/Fairness. Evaluating the bias and fairness of a system is important both from a theoretical and a practical perspective [134, 135, 136, 137]. Related studies first define a fairness criteria and then try to optimize the original objective while satisfying the fairness criteria [138, 139, 140, 141, 142, 143]. These properties are defined either at individual [138, 144, 145] or group levels [146, 139, 147].

Galhotra *et al.* [148] first applied the notion of software testing to evaluating software fairness. They mutate the sensitive features of the inputs and check whether the output changes. One major problem with their proposed method, Themis, is that it assumes the model takes into account

sensitive attribute(s) during training and inference. This assumption is not realistic since most existing fairness-aware models drop input-sensitive feature(s). Besides, Themis will not work on image classification, where the sensitive attribute (*e.g.,*, gender, race) is a visual concept that cannot be flipped easily.

Our third project(Chapter 5) proposed a white-box approach to measure the bias learned by the model during training. Our testing method does not require the model to take into account any sensitive feature(s). We propose a new fairness notion for the setting of multi-object classification, *average confusion disparity*, and a proxy, *average bias*, to measure for any deep learning model even when only unlabeled testing data is provided. In addition, our method tries to provide an explanation behind the discrimination. Our last project(Chapter 6) proposed weighted regularization methods, which can be applied in different stage of retraining or inference based on different scenarios and usage, to reduce bias among the target bias triple.

Chapter 3: DeepTest: Automated Testing of DNN based Autonomous Cars

In this chapter, we introduce our first project DeepTest, which is one of the first few papers in proposing software testing techniques for DNN based software and promoting the birth of SE for AI area. In this project, we proposed systematic test generation and automatic software testing techniques for DNN based software such as autonomous cars. We leveraged realistic image transformations to simulate different driving conditions such as camera shake, lightning condition, weather condition, *etc.* for generating corner cases and proposed neuron coverage guided test generation to synthesize corner cases and maximize the neuron coverage at the same time. We also leveraged metamorphic testing to generate oracle for these new generated test inputs to automatically test DNN based software. Finally, we implemented our proposed techniques in DeepTest and applied our tool in testing three top performing models in Udacity self-driving car challenge. Our tool identified thousands of erroneous behaviors that may lead to potential fatal crash. We also show that our generated test inputs can be used to repair those errors after retraining.

We publicly release the source code[1]. All images, figures, tables, equations, and text included in this chapter is based on a published collaborative work [53].

3.1 Introduction

Significant progress in Machine Learning (ML) techniques like Deep Neural Networks (DNNs) over the last decade has enabled the development of safety-critical ML systems like autonomous cars. Several major car manufacturers including Tesla, GM, Ford, BMW, and Waymo/Google are building and actively testing these cars. Recent results show that autonomous cars have become very efficient in practice and already driven millions of miles without any human intervention [149, 150]. Twenty US states including California, Texas, and New York have recently passed legislation

[1] https://github.com/ARiSE-Lab/deepTest

to enable testing and deployment of autonomous vehicles [151].

However, despite the tremendous progress, just like traditional software, DNN-based software, including the ones used for autonomous driving, often demonstrate incorrect/unexpected corner-case behaviors that can lead to dangerous consequences like a fatal collision. Several such real-world cases have already been reported (see Table 3.1). As Table 3.1 clearly shows, such crashes often happen under rare previously unseen corner cases. For example, the fatal Tesla crash resulted from a failure to detect a white truck against the bright sky. The existing mechanisms for detecting such erroneous behaviors depend heavily on manual collection of labeled test data or ad hoc, unguided simulation [152, 153] and therefore miss numerous corner cases. Since these cars adapt behavior based on their environment as measured by different sensors (e.g., camera, Infrared obstacle detector, etc.), the space of possible inputs is extremely large. Thus, unguided simulations are highly unlikely to find many erroneous behaviors.

Table 3.1: **Examples of real-world accidents involving autonomous cars**

	Reported Date	**Cause**	**Outcome**	**Comments**
Hyundai Competition [40]	December, 2014	Rain fall	Crashed while testing	"The sensors failed to pick up street signs, lane markings, and even pedestrians due to the angle of the car shifting in rain and the direction of the sun" [40]
Tesla autopilot mode [41]	July, 2016	Image contrast	Killed the driver	"The camera failed to recognize the white truck against a bright sky" [42]
Google self-driving car [43]	February, 2016	Failed to estimate speed	Hit a bus while shifting lane	"The car assumed that the bus would yield when it attempted to merge back into traffic" [43]

At a conceptual level, these erroneous corner-case behaviors in DNN-based software are analogous to logic bugs in traditional software. Similar to the bug detection and patching cycle in traditional software development, the erroneous behaviors of DNNs, once detected, can be fixed by adding the error-inducing inputs to the training data set and also by possibly changing the model structure/parameters. However, this is a challenging problem, as noted by large software companies like Google and Tesla that have already deployed machine learning techniques in several production-scale systems including self-driving car, speech recognition, image search, etc. [154, 155].

Our experience with traditional software has shown that it is hard to build robust safety-critical systems only using manual test cases. Moreover, the internals of traditional software and new

DNN-based software are fundamentally different. For example, unlike traditional software where the program logic is manually written by the software developers, DNN-based software automatically learns its logic from a large amount of data with minimal human guidance. In addition, the logic of a traditional program is expressed in terms of control flow statements while DNNs use weights for edges between different neurons and nonlinear activation functions for similar purposes. These differences make automated testing of DNN-based software challenging by presenting several interesting and novel research problems.

First, traditional software testing techniques for systematically exploring different parts of the program logic by maximizing branch/code coverage is not very useful for DNN-based software as the logic is not encoded using control flow [44]. Next, DNNs are fundamentally different from the models (e.g., finite state machines) used for modeling and testing traditional programs. Unlike the traditional models, finding inputs that will result in high model coverage in a DNN is significantly more challenging due to the non-linearity of the functions modeled by DNNs. Moreover, the Satisfiability Modulo Theory (SMT) solvers that have been quite successful at generating high-coverage test inputs for traditional software are known to have trouble with formulas involving floating-point arithmetic and highly nonlinear constraints, which are commonly used in DNNs. In fact, several research projects have already attempted to build custom tools for formally verifying safety properties of DNNs. Unfortunately, none of them scale well to real-world-sized DNNs [67, 156, 157]. Finally, manually creating specifications for complex DNN systems like autonomous cars is infeasible as the logic is too complex to manually encode as it involves mimicking the logic of a human driver.

In this paper, we address these issues and design a systematic testing methodology for automatically detecting erroneous behaviors of DNN-based software of self-driving cars. First, we leverage the notion of neuron coverage (i.e., the number of neurons activated by a set of test inputs) to systematically explore different parts of the DNN logic. We empirically demonstrate that changes in neuron coverage are statistically correlated with changes in the actions of self-driving cars (e.g., steering angle). Therefore, neuron coverage can be used as a guidance mechanism for systemically

exploring different types of car behaviors and identify erroneous behaviors. Next, we demonstrate that different image transformations that mimic real-world differences in driving conditions like changing contrast/brightness, rotation of the camera result in activation of different sets of neurons in the self-driving car DNNs. We show that by combining these image transformations, the neuron coverage can be increased by 100% on average compared to the coverage achieved by manual test inputs. Finally, we use transformation-specific metamorphic relations between multiple executions of the tested DNN (e.g., a car should behave similarly under different lighting conditions) to automatically detect erroneous corner case behaviors. We found thousands of erroneous behaviors across the three top performing DNNs in the Udacity self-driving car challenge [158].
The key contributions of this paper are:

- We present a systematic technique to automatically synthesize test cases that maximizes neuron coverage in safety-critical DNN-based systems like autonomous cars. We empirically demonstrate that changes in neuron coverage correlate with changes in an autonomous car's behavior.

- We demonstrate that different realistic image transformations like changes in contrast, presence of fog, etc. can be used to generate synthetic tests that increase neuron coverage. We leverage transformation-specific metamorphic relations to automatically detect erroneous behaviors. Our experiments also show that the synthetic images can be used for retraining and making DNNs more robust to different corner cases.

- We implement the proposed techniques in DeepTest, to the best of our knowledge, the first systematic and automated testing tool for DNN-driven autonomous vehicles. We use DeepTest to systematically test three top performing DNN models from the Udacity driving challenge. DeepTest found thousands of erroneous behaviors in these systems many of which can lead to potentially fatal collisions as shown in Figure 3.1.

- We have made the erroneous behaviors detected by DeepTest available at `https://deeplearningtest.github.io/deepTest/`. We also plan to release

the generated test images and the source of DeepTest for public use.

(a) original (b) with added rain

Figure 3.1: **A sample dangerous erroneous behavior found by DeepTest in the *Chauffeur* DNN.**

3.2 Methodology

To develop an automated testing methodology for DNN-driven autonomous cars we must answer the following questions. (i) How do we systematically explore the input-output spaces of an autonomous car DNN? (ii) How can we synthesize realistic inputs to automate such exploration? (iii) How can we optimize the exploration process? (iv) How do we automatically create a test oracle that can detect erroneous behaviors without detailed manual specifications? We briefly describe how DeepTest addresses each of these questions below.

3.2.1 Systematic Testing with Neuron Coverage

The input-output space (i.e., all possible combinations of inputs and outputs) of a complex system like an autonomous vehicle is too large for exhaustive exploration. Therefore, we must devise a systematic way of partitioning the space into different equivalence classes and try to cover all equivalence classes by picking one sample from each of them. In this paper, we leverage neuron coverage [44] as a mechanism for partitioning the input space based on the assumption that all inputs that have similar neuron coverage are part of the same equivalence class (i.e., the target DNN behaves similarly for these inputs).

Neuron coverage was originally proposed by Pei *et al.* for guided differential testing of multiple similar DNNs [44]. It is defined as the ratio of unique neurons that get activated for given input(s) and the total number of neurons in a DNN:

$$Neuron\ \ Coverage = \frac{|Activated\ \ Neurons|}{|Total\ \ Neurons|} \tag{3.1}$$

An individual neuron is considered activated if the neuron's output (scaled by the overall layer's outputs) is larger than a DNN-wide threshold. In this paper, we use 0.2 as the neuron activation threshold for all our experiments.

Similar to the code-coverage-guided testing tools for traditional software, DeepTest tries to generate inputs that maximize neuron coverage of the test DNN. As each neuron's output affects the final output of a DNN, maximizing neuron coverage also increases output diversity. We empirically demonstrate this effect in Section 3.4.

Pei *et al.* defined neuron coverage only for CNNs [44]. We further generalize the definition to include RNNs. Neurons, depending on the type of the corresponding layer, may produce different types of output values (*i.e.* single value and multiple values organized in a multidimensional array). We describe how we handle such cases in detail below.

For all neurons in fully-connected layers, we can directly compare their outputs against the neuron activation threshold as these neurons output a single scalar value. By contrast, neurons in convolutional layers output multidimensional feature maps as each neuron outputs the result of applying a convolutional kernel across the input space [159]. For example, the first layer in Figure 2.1.1 illustrates the application of one convolutional kernel (of size 3×3) to the entire image (5×5) that produces a feature map of size 3×3 in the succeeding layer. In such cases, we compute the average of the output feature map to convert the multidimensional output of a neuron into a scalar and compare it to the neuron activation threshold.

For RNN/LSTM with loops, the intermediate neurons are unrolled to produce a sequence of outputs (Figure 2.1.2). We treat each neuron in the unrolled layers as a separate individual neuron for the purpose of neuron coverage computation.

3.2.2 Increasing Coverage with Synthetic Images

Generating arbitrary inputs that maximize neuron coverage may not be very useful if the inputs are not likely to appear in the real-world even if these inputs potentially demonstrate buggy behaviors. Therefore, DeepTest focuses on generating realistic synthetic images by applying image transformations on seed images and mimic different real-world phenomena like camera lens distortions, object movements, different weather conditions, etc. To this end, we investigate nine different realistic image transformations (changing brightness, changing contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect). These transformations can be classified into three groups: linear, affine, and convolutional. Our experimental results, as described in Section 3.4, demonstrate that all of these transformations increase neuron coverage significantly for all of the tested DNNs. Below, we describe the details of the transformations.

Adjusting brightness and contrast are both linear transformations. The brightness of an image depends on how large the pixel values are for that image. An image's brightness can be adjusted by adding/subtracting a constant parameter β to each pixel's current value. Contrast represents the difference in brightness between different pixels in an image. One can adjust an image's contrast by multiplying each pixel's value by a constant parameter α.

Table 3.2: Different affine transformation matrices

Affine Transform	Example	Transformation Matrix	Parameters
Translation		$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	t_x: displacement along x axis t_y: displacement along y axis
Scale		$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \end{bmatrix}$	s_x: scale factor along x axis s_y: scale factor along y axis
Shear		$\begin{bmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \end{bmatrix}$	s_x: shear factor along x axis s_y: shear factor along y axis
Rotation		$\begin{bmatrix} \cos q & -\sin q & 0 \\ \sin q & \cos q & 0 \end{bmatrix}$	q: the angle of rotation

Translation, scaling, horizontal shearing, and rotation are all different types of affine transformations. An affine transformation is a linear mapping between two images that preserves points, straight lines, and planes [160]. Affine transforms are often used in image processing to fix distortions resulting from camera angle variations. In this paper, we leverage affine transformations for

the inverse case, i.e., to simulate different real-world camera perspectives or movements of objects and check how robust the self-driving DNNs are to those changes.

An affine transformation is usually represented by a 2×3 transformation matrix M [161]. One can apply an affine transformation to a 2D image matrix I by simply computing the dot product of I and M, the corresponding transformation matrix. We list the transformation matrices for the four types of affine transformations (translation, scale, shear, and rotation) used in this paper in Table 3.2.

Blurring and adding fog/rain effects are all convolutional transformations, i.e., they perform the convolution operation on the input pixels with different transform-specific kernels. A convolution operation adds (weighted by the kernel) each pixel of the input image to its local neighbors. We use four different types of blurring filters: averaging, Gaussian, median, and bilateral [162]. We compose multiple filters provided by Adobe Photoshop on the input images to simulate realistic fog and rain effects [163, 164].

3.2.3 Combining Transformations to Increase Coverage

As the individual image transformations increase neuron coverage, one obvious question is whether they can be combined to further increase the neuron coverage. Our results demonstrate that different image transformations tend to activate different neurons, i.e., they can be stacked together to further increase neuron coverage. However, the state space of all possible combinations of different transformations is too large to explore exhaustively. We provide a neuron-coverage-guided greedy search technique for efficiently finding combinations of image transformations that result in higher coverage (see Algorithm 1).

Algorithm 1: Greedy search for combining image transformations to increase neuron coverage

```
Input  : Transformations T, Seed images I
Output : Synthetically generated test images
Variable: S: stack for storing newly generated images
          Tqueue: transformation queue
1
2  Push all seed imgs ∈ I to Stack S
3  genTests = φ
4  while S is not empty do
5      img = S.pop()
6      Tqueue = φ
7      numFailedTries = 0
8      while numFailedTries ≤ maxFailedTries do
9          if Tqueue is not empty then
10             T1 = Tqueue.dequeue()
11         else
12             Randomly pick transformation T1 from T
13         end
14         Randomly pick parameter P1 for T1
15         Randomly pick transformation T2 from T
16         Randomly pick parameter P2 for T2
17         newImage = ApplyTransforms(image, T1, P1, T2, P2)
18         if covInc(newimage) then
19             Tqueue.enqueue(T1)
20             Tqueue.enqueue(T2)
21             UpdateCoverage()
22             genTest = genTests ∪ newimage S.push(newImage)
23         else
24             numFailedTries = numFailedTries + 1
25         end
26     end
27 end
28 return genTests
```

The algorithm takes a set of seed images I, a list of transformations T and their corresponding parameters as input. The key idea behind the algorithm is to keep track of the transformations that successfully increase neuron coverage for a given image and prioritize them while generating more synthetic images from the given image. This process is repeated in a depth-first manner to all images.

3.2.4 Creating a Test Oracle with Metamorphic Relations

One of the major challenges in testing a complex DNN-based system like an autonomous vehicle is creating the system's specifications manually, against which the system's behavior can

be checked. It is challenging to create detailed specifications for such a system as it essentially involves recreating the logic of a human driver. To avoid this issue, we leverage metamorphic relations [165] between the car behaviors across different synthetic images. The key insight is that even though it is hard to specify the correct behavior of a self-driving car for every transformed image, one can define relationships between the car's behaviors across certain types of transformations. For example, the autonomous car's steering angle should not change significantly for the same image under any lighting/weather conditions, blurring, or any affine transformations with small parameter values. Thus, if a DNN model infers a steering angle θ_o for an input seed image I_o and a steering angle θ_t for a new synthetic image I_t, which is generated by applying the transformation t on I_o, one may define a simple metamorphic relation where θ_o and θ_t are identical.

However, there is usually no single correct steering angle for a given image, i.e., a car can safely tolerate small variations. Therefore, there is a trade-off between defining the metamorphic relations very tightly, like the one described above (may result in a large number of false positives) and making the relations more permissive (may lead to many false negatives). In this paper, we strike a balance between these two extremes by using the metamorphic relations defined below.

To minimize false positives, we relax our metamorphic relations and allow variations within the error ranges of the original input images. We observe that the set of outputs predicted by a DNN model for the original images, say $\{\theta_{o1}, \theta_{o2}, ..., \theta_{on}\}$, in practice, result in a small but non-trivial number of errors *w.r.t.* their respective manual labels ($\{\hat{\theta}_1, \hat{\theta}_2, ..., \hat{\theta}_n\}$). Such errors are usually measured using Mean Squared Error (MSE), where $MSE_{orig} = \frac{1}{n}\sum_{i=1}^{n}(\hat{\theta}_i - \theta_{oi})^2$. Leveraging this property, we redefine a new metamorphic relation as:

$$(\hat{\theta}_i - \theta_{ti})^2 \leq \lambda \; MSE_{orig} \tag{3.2}$$

The above equation assumes that the errors produced by a model for the transformed images as input should be within a range of λ times the MSE produced by the original image set. Here, λ is a configurable parameter that allows us to strike a balance between the false positives and false

negatives.

3.3 Implementation

Autonomous driving DNNs. We evaluate our techniques on three DNN models that won top positions in the Udacity self-driving challenge [158]: Rambo [166] (2^{nd} rank), Chauffeur [167] (3^{rd} rank), and Epoch [168] (6^{th} rank). We choose these three models as their implementations are based on the Keras framework [169] that our current prototype of DeepTest supports. The details of the DNN models and dataset are summarized in Table 3.3.

As shown in the right figure of Table 3.3, the steering angle is defined as the rotation degree between the heading direction of the vehicle (the vertical line) and the heading directions of the steering wheel axles (*i.e.*, usually front wheels). The negative steering angle indicates turning left while the positive values indicate turning left. The maximum steering angle of a car varies based on the hardware of different cars. The Udacity self-driving challenge dataset used in this paper has a maximum steering angle of +/- 25 degree [158]. The steering angle is then scaled by 1/25 so that the prediction should fall between -1 and 1.

Model	Sub-Model	No. of Neurons	Reported MSE	Our MSE
Chauffeur	CNN LSTM	1427 513	0.06	0.06
Rambo	S1(CNN) S2(CNN) S3(CNN)	1625 3801 13473	0.06	0.05
Epoch	CNN	2500	0.08	0.10

† dataset HMB_3.bag [170]

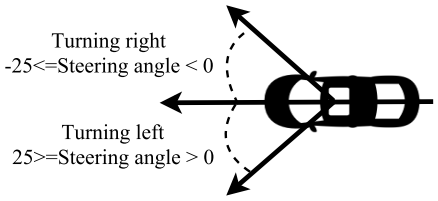


Table 3.3: (Left) Details of DNNs used to evaluate DeepTest.†(Right) The outputs of the DNNs are the steering angles for a self-driving car heading forward. The Udacity self-driving car has a maximum steering angle of +/- 25 degree.

Rambo model consists of three CNNs whose outputs are merged using a final layer [166]. Two of the CNNs are inspired by NVIDIA's self-driving car architecture [91], and the third CNN is based on comma.ai's steering model [171]. As opposed to other models that take individual images as input, Rambo takes the differences among three consecutive images as input. The model uses Keras [169] and Theano [172] frameworks.

Chauffeur model includes one CNN model for extracting features from the image and one LSTM model for predicting steering angle [167]. The input of the CNN model is an image while the input of the LSTM model is the concatenation of 100 features extracted by the CNN model from previous 100 consecutive images. Chauffeur uses Keras [169] and Tensorflow [173] frameworks.

Epoch model uses a single CNN. As the pre-trained model for Epoch is not publicly available, we train the model using the instructions provided by the authors [168]. We used the CH2_002 dataset [170] from the Udacity self-driving Challenge for training the epoch model. Epoch , similar to Chauffeur, uses Keras and Tensorflow frameworks.

Image transformations. In the experiments for RQ2 and RQ3, we leverage seven different types of simple image transformations: translation, scaling, horizontal shearing, rotation, contrast adjustment, brightness adjustment, and blurring. We use OpenCV to implement these transformations [174]. For RQ2 and RQ3 described in Section 3.4, we use 10 parameters for each transformation as shown in Table 3.4.

Table 3.4: Transformations and parameters used by DeepTest for generating synthetic images.

Transformations		Parameters	Parameter ranges
Translation		(t_x, t_y)	(10, 10) to (100, 100) step (10, 10)
Scale		(s_x, s_y)	(1.5, 1.5) to (6, 6) step (0.5, 0.5)
Shear		(s_x, s_y)	$(-1.0, 0)$ to $(-0.1, 0)$ step $(0.1, 0)$
Rotation		q (degree)	3 to 30 with step 3
Contrast		α (gain)	1.2 to 3.0 with step 0.2
Brightness		β (bias)	10 to 100 with step 10
Blur	**Averaging**	kernel size	3×3, 4×4, 5×5, 6×6
	Gaussian	kernel size	3×3, 5×5, 7×7 , 3×3
	Median	aperture linear size	3, 5
	Bilateral Filter	diameter, sigmaColor, sigmaSpace	9, 75, 75

3.4 Results

As DNN-based models are fundamentally different than traditional software, first, we check whether neuron coverage is a good metric to capture functional diversity of DNNs. In particular, we investigate whether neuron coverage changes with different input-output pairs of an autonomous

car. An individual neuron's output goes through a sequence of linear and nonlinear operations before contributing to the final outputs of a DNN. Therefore, it is not very clear how much (if at all) individual neuron's activation will change the final output. We address this in our first research question.

Table 3.5: Relation between neuron coverage and test output

Model	Sub-Model	Steering Angle	Steering Direction	
		Spearman Correlation	Wilcoxon Test	Effect size (Cohen's d)
Chauffeur	Overall	-0.10 (***)	left (+ve) > right (-ve) (***)	negligible
	CNN	0.28 (***)	left (+ve) < right (-ve) (***)	negligible
	LSTM	-0.10 (***)	left (+ve) > right (-ve) (***)	negligible
Rambo	Overall	-0.11 (***)	left (+ve) < right (-ve) (***)	negligible
	S1	-0.19 (***)	left (+ve) < right (-ve) (***)	large
	S2	0.10 (***)	not significant	negligible
	S3	-0.11 (***)	not significant	negligible
Epoch	N/A	0.78 (***)	left (+ve) < right (-ve) (***)	small

*** indicates statistical significance with p-value $< 2.2 * 10^{-16}$

RQ1. Do different input-output pairs result in different neuron coverage?

For each input image we measure the neuron coverage (see Equation 3.1 in Section 3.2.1) of the underlying models and the corresponding output. As discussed in Section 3.3, corresponding to an input image, each model outputs a steering direction (left (+ve) / right (-ve)) and a steering angle as shown in Table 3.3 (right). We analyze the neuron coverage for both of these outputs separately.

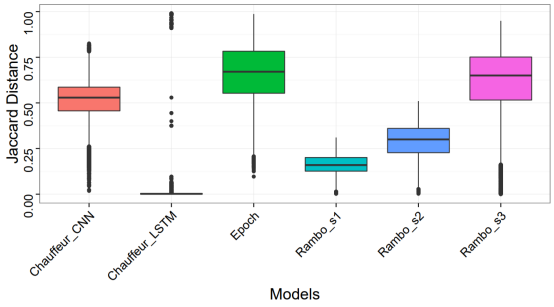
Steering angle. As steering angle is a continuous variable, we check Spearman rank correlation [175] between neuron coverage and steering angle. This is a non-parametric measure to compute monotonic association between the two variables [176]. Correlation with positive statistical significance suggests that the steering angle increases with increasing neuron coverage and vice versa. Table 3.5 shows that Spearman correlations for all the models are statistically significant—while Chauffeur and Rambo models show an overall negative association, Epoch model shows a strong positive correlation. This result indicates that the neuron coverage changes with the changes in output steering angles, *i.e.* different neurons get activated for different outputs. Thus, in this setting, neuron coverage can be a good approximation for estimating the diversity of input-output pairs. Moreover, our finding that monotonic correlations between neuron coverage

and steering angle also corroborate Goodfellow et al.'s hypothesis that, in practice, DNNs are often highly linear [113].

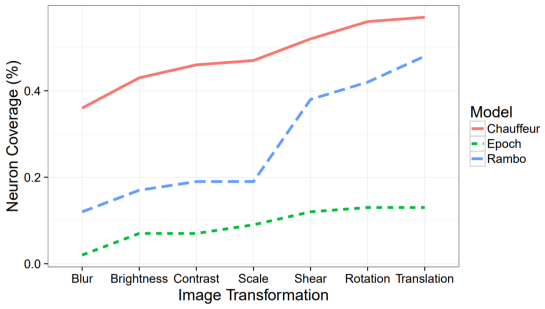
Steering direction. To measure the association between neuron coverage and steering direction, we check whether the coverage varies between right and left steering direction. We use the Wilcoxon nonparametric test as the steering direction can only have two values (left and right). Our results confirm that neuron coverage varies with steering direction with statistical significance ($p < 2.2 * 10^{-16}$) for all the three overall models. Interestingly, for Rambo , only the Rambo-S1 sub-model shows statistically significant correlation but not Rambo-S2 and Rambo-S3. These results suggest that, unlike steering angle, some sub-models are more responsible than other for changing steering direction.

Overall, these results show that neuron coverage altogether varies significantly for different input-output pairs. Thus, a neuron-coverage-directed (NDG) testing strategy can help in finding corner cases.

Result 1: *Neuron coverage is correlated with input-output diversity and can be used to systematic test generation.*



4.1 Difference in neuron coverage caused by different image transformations



4.2 Average cumulative neuron coverage per input image

Figure 3.2: Different image transformations activate significantly different neurons. In the top figure the median Jaccard distances for Chauffeur-CNN, Chauffeur-LSTM, Epoch, Rambo-S1, Rambo-S2, and Rambo-S3 models are 0.53, 0.002, 0.67, 0.12, 0.17, 0.30, and 0.65.

Next, we investigate whether synthetic images generated by applying different realistic image transformations (as described in Table 3.2) on existing input images can activate different neurons.

Thus, we check:

RQ2. Do different realistic image transformations activate different neurons?

We randomly pick 1,000 input images from the test set and transform each of them by using seven different transformations: blur, brightness, contrast, rotation, scale, shear, and translation. We also vary the parameters of each transformation and generate a total of 70,000 new synthetic images. We run all models with these synthetic images as input and record the neurons activated by each input.

We then compare the neurons activated by different synthetic images generated from the same image. Let us assume that two transformations T_1 and T_2, when applied to an original image I, activate two sets of neurons N_1 and N_1 respectively. We measure the dissimilarities between N_1 and N_2 by measuring their Jaccard distance: $1 - \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|}$.

Figure 3.2.1 shows the result for all possible pair of transformations (*e.g.,* blur vs. rotation, rotation vs. transformation, *etc.*) for different models. These results indicate that for all models, except Chauffeur-LSTM , different transformations activate different neurons. As discussed in Section 2.2.2, LSTM is a particular type of RNN architecture that keeps states from previous inputs and hence increasing the neuron coverage of LSTM models with single transformations is much harder than other models. In this paper, we do not explore this problem any further and leave it as an interesting future work.

We further check how much a single transformation contributes in increasing the neuron coverage *w.r.t.* all other transformations for a given seed image. Thus, if an original image I undergoes seven discrete transformations: $T_1, T_2, ...T_7$, we compute the total number of neurons activated by $T_1, T_1 \cup T_2, ..., \bigcup_{i=1}^{7} T_i$. Figure 3.2.2 shows the cumulative effect of all the transformations on average neuron coverage per seed image. We see that the cumulative coverage increases with increasing number of transformations for all the models. In other words, all the transformations are contributing towards the overall neuron coverage.

We also compute the percentage change in neuron coverage per image transformation (N_T)

w.r.t. to the corresponding seed image (N_O) as: (N_T-N_O)/N_O. Figure 3.3 shows the result. For all the studied models, the transformed images increase the neuron coverage significantly—Wilcoxon nonparametric test confirms the statistical significance. These results also show that different image transformations increase neuron coverage at different rates.

Result 2: *Different image transformations tend to activate different sets of neurons.*

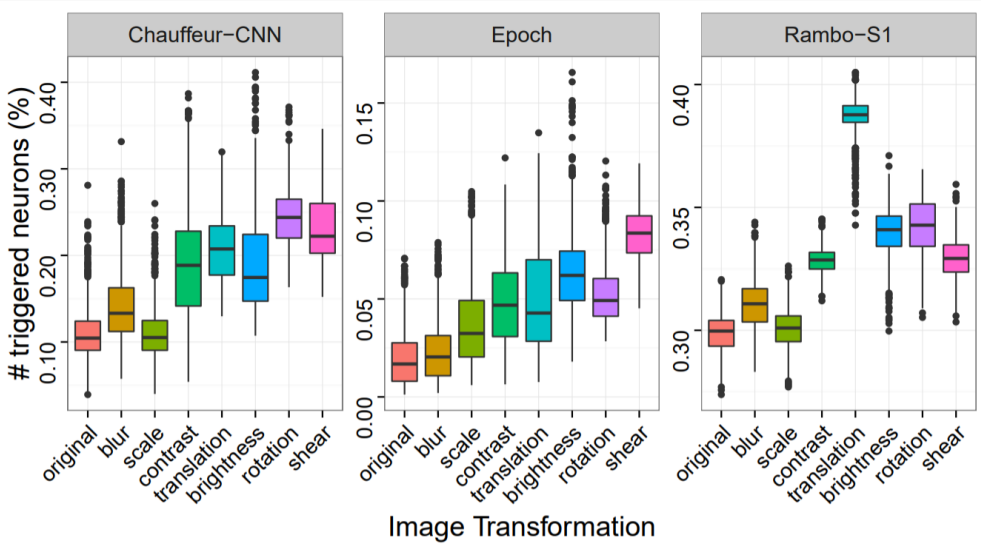
Next, we mutate the seed images with different combinations of transformations (see Section 3.2). Since different image transformations activate different set of neurons, here we try to increase the neuron coverage by these transformed image inputs. To this end, we question:

RQ3. Can neuron coverage be further increased by combining different image transformations?

We perform this experiment by measuring neuron coverage in two different settings: (i) applying a set of transformations and (ii) combining transformations using *coverage-guided* search.

i) *Cumulative Transformations.* Since different seed images activate a different set of neurons (see RQ1), multiple seed images collectively achieve higher neuron coverage than a single one. Hence, we check whether the transformed images can still increase the neuron coverage collectively *w.r.t.* the cumulative baseline coverage of a set of seed images. In particular, we generate a total of 7,000 images from 100 seed images by applying 7 transformations and varying 10 parameters on 100 seed images. This results in a total of 7,000 test images. We then compare the cumulative neuron coverage of these synthetic images *w.r.t.* the baseline, which use the same 100 seed images for fair comparison. Table 3.6 shows the result. Across all the models (except Rambo-S3), the cumulative coverage increased significantly. Since the Rambo-S3 baseline already achieved 97% coverage, the transformed images only increase the coverage by $(13,080 - 13,008)/13,008 = 0.55\%$.

ii) *Guided Transformations.* Finally, we check whether we can further increase the cumulative neuron coverage by using the coverage-guided search technique described in Algorithm 1. We generate 254, 221, and 864 images from 100 seed images for Chauffeur-CNN , Epoch , and



Median Increase in Neuron Coverage

Transformation	Chauffeur (CNN,LSTM)	Epoch	Rambo (S1,S2,S3)
Scale	(1.0,0.0) (0.67%,0%)	39.0** 93%	(2.0*,5.0*,32.0) (0.41%,1%,4%)
Brightness	(100.0**,1.0) (67%,0.2%)	113.0** 269%	(67.0**,104.0**,585.0*) (14%,24%,66%)
Contrast	(120.0**,1.0*) (80%,0.2%)	75.0** 179%	(47.0**,100.0**,159.0) (10%,23%,18%)
Blur	(41.0**,0.0) (28%,0%)	9.0* 21%	(18.0**,23.0**,269.5*) (4%,5%,31%)
Rotation	(199.0**,2.0*) (134%,0.39%)	81.0** 193%	(70.0**,13.0**,786.5*) (14%,3%,89%)
Translation	(147.0**,1.0*) (99%,0.2%)	65.0** 155%	(143.0**,167.0**,2315.5**) (29%,38%,263%)
Shear	(168.0**,1.0*) (113%,0.2%)	167.0** 398%	(48.0**,132.0**,1472.0**) (10%,30%,167%)

All numbers are statistically significant;
Numbers with * and ** have small and large Cohen's D effect.

Figure 3.3: **Neuron coverage per seed image for individual image transformations *w.r.t.* baseline.**

Table 3.6: Neuron coverage achieved by cumulative and guided transformations applied to 100 seed images.

Model	Baseline	Cumulative Transformation	Guided Generation	% increase of guided *w.r.t.* Baseline	Cumulative
Chauffeur-CNN	658 (46%)	1,065 (75%)	1,250 (88%)	90%	17%
Epoch	621 (25%)	1034 (41%)	1,266 (51%)	104%	22%
Rambo-S1	710 (44%)	929 (57%)	1,043 (64%)	47%	12%
Rambo-S2	1,146 (30%)	2,210 (58%)	2,676 (70%)	134%	21%
Rambo-S3	13,008 (97%)	13,080 (97%)	13,150 (98%)	1.1%	0.5%

Rambo models respectively and measure their collective neuron coverage. As shown in Table 3.6, the guided transformations collectively achieve 88%, 51%, 64%, 70%, and 98% of total neurons for models Chauffeur-CNN , Epoch , Rambo-S1 , Rambo-S2 , and Rambo-S3 respectively, thus increasing the coverage up to 17% 22%, 12%, 21%, and 0.5% *w.r.t.* the unguided approach. This method also significantly achieves higher neuron coverage *w.r.t.* baseline cumulative coverage.

Result 3: *By systematically combining different image transformations, neuron coverage can be improved by around 100% w.r.t. the coverage achieved by the original seed images.*

Next we check whether the synthetic images can trigger any erroneous behavior in the autonomous car DNNs and if we can detect those behaviors using metamorphic relations as described in Section 3.2.4. This leads to the following research question:

RQ4. Can we automatically detect erroneous behaviors using metamorphic relations?

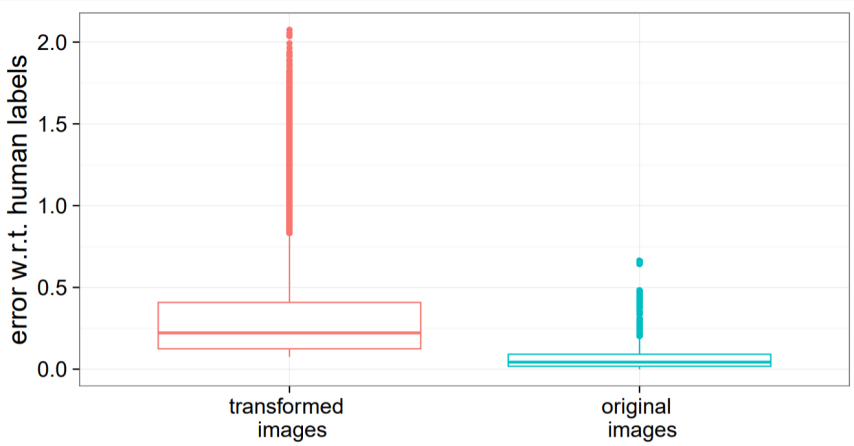


Figure 3.4: **Deviations from the human labels for images that violate the metamorphic relation (see Equation 3.2) is higher compared to the deviations for original images. Thus, these synthetic images have a high chance to show erroneous behaviors.**

Here we focus on the transformed images whose outputs violate the metamorphic relation defined in Equation 3.2. We call these images I_{err} and their corresponding original images as I_{org}. We compare the deviation between the outputs of I_{err} and I_{org} *w.r.t.* the corresponding human labels, as shown in Figure 3.4. The deviations produced for I_{err} are much larger than I_{org} (also

confirmed by Wilcoxon test for statistical significance). In fact, mean squared error (MSE) for I_{err} is 0.41, while the MSE of the corresponding I_{org} is 0.035. Such differences also exist for other synthetic images that are generated by composite transformations including rain, fog, and those generated during the coverage-guided search. Thus, overall I_{err} has a higher potential to show buggy behavior.

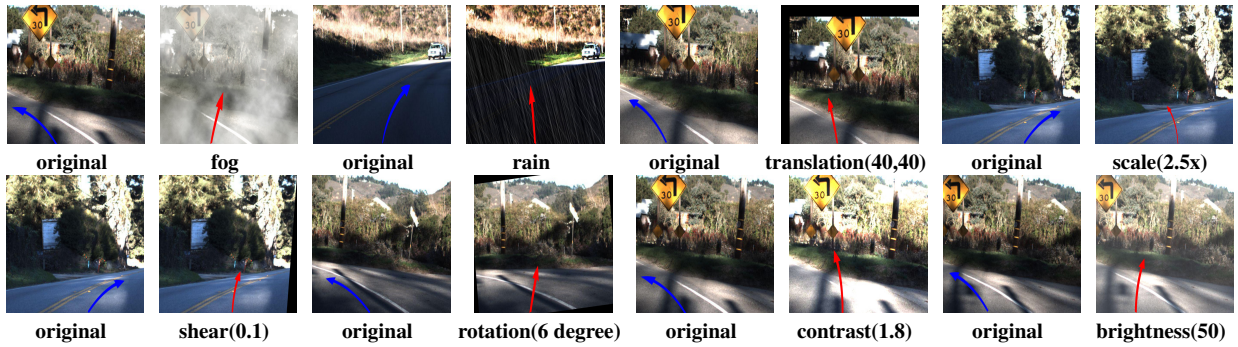


Figure 3.5: Sample images showing erroneous behaviors detected by DeepTest using synthetic images. For **original** images the arrows are marked in blue, while for the synthetic images they are marked in **red**. More such samples can be viewed at `https://deeplearningtest.github.io/deepTest/`.

However, for certain transformations (e.g., rotation), not all violations of the metamorphic relations can be considered buggy as the correct steering angle can vary widely based on the contents of the transformed image. For example, when an image is rotated by a large amount, say 30 degrees, it is nontrivial to automatically define its correct output behavior without knowing its contents. To reduce such false positives, we only report bugs for the transformations (e.g., small rotations, rain, fog, etc.) where the correct output should not deviate much from the labels of the corresponding seed images. Thus, we further use a filtration criteria as defined in Equation 3.3 to identify such transformations by checking whether the MSE of the synthetic images is close to that of the original images.

$$| MSE_{(trans,param)} - MSE_{org} | \leq \epsilon \tag{3.3}$$

Thus, we only choose the transformations that obey Equation 3.3 for counting erroneous behaviors. Table 3.7 shows the number of such erroneous cases by varying two thresholds: ϵ and

λ—a higher value of λ and lower value of ϵ makes the system report fewer bugs and vice versa. For example, with a λ of 5 and ϵ of 0.03, we report 330 violations for simple transformations. We do not enforce the filtration criteria for composite transformations. Rain and fog effects should produce same outputs as original images. Also, in guided search since multiple transformations produce the synthesized images, it is not possible to filter out a single transformation. Thus, for rain, fog, and guided search, we report 4448, 741, and 821 erroneous behavior respectively for λ = 5, across all three models.

Table 3.7: Number of erroneous behaviors reported by DeepTest across all tested models at different thresholds

λ (see Eqn. 3.2)	Simple Tranformation ϵ (see Eqn. 3.3) 0.01	0.02	0.03	0.04	0.05	Composite Transformation Fog	Rain	Guided Search
1	15666	18520	23391	24952	29649	9018	6133	1148
2	4066	5033	6778	7362	9259	6503	2650	1026
3	1396	1741	2414	2627	3376	5452	1483	930
4	501	642	965	1064	4884	4884	997	872
5	95	171	**330**	382	641	**4448**	**741**	**820**
6	49	85	185	210	359	4063	516	764
7	13	24	89	105	189	3732	287	721
8	3	5	34	45	103	3391	174	668
9	0	1	12	19	56	3070	111	637
10	0	0	3	5	23	2801	63	597

Table 3.8: Number of unique erroneous behaviors reported by DeepTest for different models with λ = 5 (see Eqn. 3.2)

Transformation	Chauffeur	Epoch	Rambo
Simple Transformation			
Blur	3	27	11
Brightness	97	32	15
Contrast	31	12	-
Rotation	-	13	-
Scale	-	10	-
Shear	-	-	23
Translation	21	35	-
Composite Transformation			
Rain	650	64	27
Fog	201	135	4112
Guided	89	65	666

Table 3.8 further elaborates the result for different models for λ = 5 and ϵ = 0.03, as highlighted in Table 3.7. Interestingly, some models are more prone to erroneous behaviors for some transformations than others. For example, Rambo produces 23 erroneous cases for shear, while the other two models do not show any such cases. Similarly, DeepTest finds 650 instances of erroneous behavior in Chauffeur for rain but only 64 and 27 for Epoch and Rambo respectively. In total, DeepTest detects 6339 erroneous behaviors across all three models. Figure 3.5 further shows

some of the erroneous behaviors that are detected by DeepTest under different transformations that can lead to potentially fatal situations. We also manually checked the bugs reported in Table 3.8 and report the false positives in Figure 3.6. It also shows two synthetic images (the corresponding original images) where DeepTest incorrectly reports erroneous behaviors while the model's output is indeed safe. Although such manual investigation is, by definition, subjective and approximate, all the authors have reviewed the images and agreed on the false positives.

Model	Simple Transformation	Guided	Rain	Fog	Total
Epoch	14	0	0	0	14
Chauffeur	5	3	12	6	26
Rambo	8	43	11	28	90
Total	27	46	23	34	130

original **translation(50,50), epoch** **original** **shear(0.4), rambo**

Figure 3.6: Sample false positives produced by DeepTest for $\lambda = 5$, $\epsilon = 0.03$

Result 4: *With neuron coverage guided synthesized images, DeepTest successfully detects more than 1,000 erroneous behavior as predicted by the three models with low false positives.*

RQ5. Can retraining DNNs with synthetic images improve accuracy?

Table 3.9: Improvement in MSE after retraining of Epoch model with synthetic tests generated by DeepTest

Test set	Original MSE	Retrained MSE
original images	0.10	0.09
with fog	0.18	0.10
with rain	0.13	0.07

Here we check whether retraining the DNNs with some of the synthetic images generated by DeepTest helps in making the DNNs more robust.We used the images from HMB_3.bag [170] and created their synthetic versions by adding the rain and fog effects. We retrained the Epoch model with randomly sampled 66% of these synthetic inputs along with the original training data. We evaluated both the original and the retrained model on the rest 34% of the synthetic images and their original versions. In all cases, the accuracy of the retrained model improved significantly over the original model as shown in Table 3.9.

Result 5: *Accuracy of a DNN can be improved up to 46% by retraining the DNN with synthetic data generated by DeepTest.*

3.5 Discussion & Threats to Validity

DeepTest generates realistic synthetic images by applying different image transformations on the seed images. However, these transformations are not designed to be exhaustive and therefore may not cover all realistic cases.

While our transformations like rain and fog effects are designed to be realistic, the generated pictures may not be exactly reproducible in reality due to a large number of unpredictable factors, e.g., the position of the sun, the angle and size of the rain drops. etc. However, as the image processing techniques become more sophisticated, the generated pictures will get closer to reality.

A complete DNN model for driving an autonomous vehicle must also handle braking and acceleration besides the steering angle. We restricted ourselves to only test the accuracy of the steering angle as our tested models do not support braking and acceleration yet. However, our techniques should be readily applicable to testing those outputs too assuming that the models support them.

3.6 Related Work

Testing of driver assistance systems. Abdessalem et al. proposed a technique for testing Advanced Driver Assistance Systems (ADAS) in autonomous cars that show warnings to the drivers if it detects pedestrians in positions with low driver visibility [177]. They use multi-objective meta heuristic search algorithms to generate tests that simultaneously focus on the most critical behaviors of the system and the environment as decided by the domain experts (*e.g.,* moving pedestrians in the dark).

The key differences between this work and ours are threefold: (i) We focus on testing the image recognition and steering logic in the autonomous car DNNs while their technique tested

ADAS system's warning logic based on preprocessed sensor inputs; (ii) Their blackbox technique depends on manually selected critical scenarios while our gray-box technique looks inside the DNN model and systematically maximize neuron coverage. The trade-off is that their technique can, in theory, work for arbitrary implementations while our technique is tailored for DNNs; and (iii) We leverage metamorphic relations for creating a test oracle while they depend on manual specifications for detecting faulty behavior.

Testing and verification of machine learning. Traditional practices in evaluating machine learning systems primarily measure their accuracy on randomly drawn test inputs from manually labeled datasets and ad hoc simulations [178, 153, 152]. However, without the knowledge of the model's internals, such blackbox testing paradigms are not able to find different corner-cases that may induce unexpected behaviors [179, 44].

Pei *et al.* [44] proposed DeepXplore, a whitebox differential testing algorithm for systematically finding inputs that can trigger inconsistencies between multiple DNNs. They introduced neuron coverage as a systematic metric for measuring how much of the internal logic of a DNNs have been tested. By contrast, our graybox methods use neuron coverage for guided test generation in a single DNN and leverage metamorphic relations to identify erroneous behaviors without requiring multiple DNNs.

Another recent line of work has explored the possibility of verifying DNNs against different safety properties [157, 156, 67]. However, none of these techniques can verify a rich set of properties for real-world-sized DNNs. By contrast, our techniques can systematically test state-of-the-art DNNs for safety-critical erroneous behaviors but do not provide any theoretical guarantees.

Adversarial machine learning.

A large number of projects successfully attacked machine learning models at test time by forcing it to make unexpected mistakes. More specifically, these attacks focus on finding inputs that, when changed minimally from their original versions, get classified differently by the machine learning classifiers. These types of attacks are known to affect a broad spectrum of tasks such as image recognition [119, 113, 115, 116, 180, 118, 117, 114, 77], face detection/verification [181,

182], malware detection [183, 184, 185, 186], and text analysis [187, 188]. Several prior works have explored defenses against these attacks with different effectiveness [126, 130, 122, 132, 156, 123, 189, 190, 129, 131, 125, 191, 124, 127, 128].

In summary, this line of work tries to find a particular class of erroneous behaviors, i.e., forcing incorrect prediction by adding a minimum amount of noise to a given input. By contrast, we systematically test a given DNN by maximizing neuron coverage and find a diverse set of corner-case behaviors. Moreover, we specifically focus on finding realistic conditions that can occur in practice.

Test amplification. There is a large body of work on test case generation and amplification techniques for traditional software that automatically generate test cases from some seed inputs and increase code coverage. Instead of summarizing them individually here, we refer the interested readers to the surveys by Anand et al. [87], McMinn et al. [9], and Pasareanu et al. [88]. Unlike these approaches, DeepTest is designed to operate on DNNs.

Metamorphic testing. Metamorphic testing [165, 192] is a way of creating test oracles in settings where manual specifications are not available. Metamorphic testing identifies buggy behavior by detecting violations of domain-specific metamorphic relations that are defined across outputs from multiple executions of the test program with different inputs. For example, a sample metamorphic property for program p adding two inputs a and b can be $p(a, b) = p(a, 0) + p(b, 0)$. Metamorphic testing has been used in the past for testing both supervised and unsupervised machine learning classifiers [193, 20]. By contrast, we define new metamorphic relations in the domain of autonomous cars which, unlike the classifiers tested before, produce a continuous steering angle, i.e., it is a regression task.

3.7 Conclusion

In this paper, we proposed and evaluated DeepTest, a tool for automated testing of DNN-driven autonomous cars. DeepTest maximizes the neuron coverage of a DNN using synthetic test images generated by applying different realistic transformations on a set of seed images. We use

domain-specific metamorphic relations to find erroneous behaviors of the DNN without detailed specification. DeepTest can be easily adapted to test other DNN-based systems by customizing the transformations and metamorphic relations. We believe DeepTest is an important first step towards building robust DNN-based systems.

Chapter 4: Understanding Local Robustness of Deep Neural Networks under Natural Variations

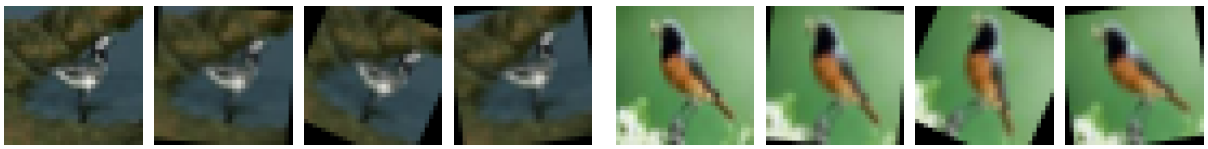
In Chapter 3, DeepTest has identified large numbers of erroneous behaviors resulted from corner cases generated by natural transformations. In this chapter, we presented a follow-up project DeepRobust in studying the local robustness of deep neural networks under natural variations. We first conducted an empirical study to understand per-point robustness of deep neural networks under natural variation. We found that not all the inputs under natural variation will result in erroneous outputs for a DNN. There exists specific weak data points, which are more likely to fail a deep neural network model than other data points. Then we designed and implemented a white-box approach(DEEPROBUST-W) and a black-box approach(DEEPROBUST-B) to identify these weak data points for DNN based image classifiers and DNN based self-driving cars. We evaluated our approaches on 9 DNN based image classifiers and 3 DNN based self-driving car models. Our results show that DEEPROBUST-W and DEEPROBUST-B are able to achieve an F1 score of up to 91.4% and 99.1%, respectively in testing DNN based image classifiers. DEEPROBUST-W is effective in identifying weak data points with F1 score up to 78.9% in testing DNN based self-driving car models.

We publicly release the source code[1]. All images, figures, tables, equations, and text included in this chapter is based on a published collaborative work [76].

4.1 Introduction

Deep Neural Networks (DNNs) have achieved an unprecedented level of performance over the last decade in many sophisticated areas such as image recognition [92], self-driving cars [91] and

[1] https://github.com/deeprobust/DeepRobust



(a) 0°, bird (b) +6°, airplane (c) +24°, cat (d) -9°, dog (e) 0°, bird (f) +6°, bird (g) +24°, bird (h) -9°, bird

Figure 4.1: **(a)-(d) A well-trained Resnet model [47] misclassifies the rotated variations of a bird image into three different classes though the original un-rotated image is classified correctly. (e)-(h) The same model successfully classifies all the rotated variants of another bird image from the same test set. The sub-captions consist of rotation degrees and the predicted classes.**

playing complex games [194]. These advances have also motivated companies to adapt their software development flows to incorporate AI components [195]. This trend has, in turn, spawned a new area of research within software engineering addressing the quality assurance of DNN components [44, 45, 46, 65, 196, 197, 198, 199, 200, 201, 78, 55].

Notwithstanding the impressive capabilities of DNNs, recent research has shown that DNNs can be easily fooled, *i.e.* made to mispredict, with a little variation of the input data [113, 47, 45]—either adding a norm-bound pixel-level perturbation into the original input [119, 113, 123], or with *natural* variants of the inputs, *e.g.,* rotating an image, changing the lighting conditions, *etc.* [47, 44]. The natural variants are especially concerning as they can occur naturally in the field without any active adversary and may lead to serious consequences [45, 46].

While norm-bound perturbation based DNN robustness is relatively well-studied, our knowledge of DNN robustness under the natural variations is still limited—we do not know which images are more robust than others, what their characteristics are, etc. For example, consider Figure 4.1: although the original bird image (a) is predicted correctly by a DNN, its rotated variations in images (b)-(d) are mispredicted to three different classes. This makes the original image (a) very weak as far as robustness is concerned. In contrast, the bird image (e) and all its rotated versions (generated by the same degrees of rotation) in Figure 4.1:(f)-(h) are correctly classified. Thus, the original image (e) is quite robust. It is important to distinguish between such robust vs. non-robust images, as the non-robust ones can induce errors with slight natural variations.

Existing literature, however, focuses on estimating the overall robustness of DNNs across all

the test data [47, 48, 49]. From a traditional software point of view, this is analogous to estimating how buggy a software is without actually localizing the bugs. Our current work tries to bridge this gap by localizing the non-robust points in the input space that pose significant threats to a DNN model's robustness. However, unlike traditional software where bug localization is performed in program space, we identify the non-robust inputs in the data space. As a DNN is a combination of data and architecture, and the architecture is largely uninterpretable, we restrict our study of non-robustess to the input space. To this end, we first quantify the local (per input) robustness property of a DNN. First, we treat all the natural variants of an input image as its *neighbors*. Then, for each input data, we consider a population of its neighbors and measure the fraction of this population classified correctly by the DNN - a high fraction of correct classifications indicates good robustness (Figure 4.1:e) and vice versa (Figure 4.1:a). We term this measure *neighbor accuracy*. Using this metric, we study different local robustness properties of the DNNs and analyze how the weak, *a.k.a.* non-robust, points differ characteristically from their robust counterparts. Given that the number of natural neighbors of an image can be potentially infinite, first we performed a more controlled analysis by keeping the natural variants limited to spatially transformed images generated by rotation and translation, following the previous work [47, 48, 49]. Such controlled experiments help us to explore different robustness properties while systematically varying transformation parameters.

Our analysis with three well-known object recognition datasets across three popular DNN models, *i.e.* a total of nine DNN-dataset combinations, reveal several interesting properties of local robustness of a DNN *w.r.t.* natural variants:

- The neighbors of a weaker point are not necessarily classified to one single incorrect class. In fact, the weaker the point is its neighbors (mis)classifications become more diverse.

- The weak points are concentrated towards the class decision boundaries of the DNN in the feature space.

Based on these findings, we further develop two techniques (a black-box and a white-box) that can localize the points of poor robustness, thereby providing a means of, input-specific, real-time

feedback about robustness to the end-user. Our white-box and black-box detectors can identify weak, *a.k.a.* non-robust, points with f1 score up to 91.4% and 99.1%, respectively, at neighbor accuracy cutoff 0.75. To further check the generalizability of our technique, we aim to detect weak points *w.r.t.* a self-driving car application where we generated natural input variants by adding rain and fog. Note that these are more complex image transformations, and also the model works in a regression setting instead of classification. These models take an image as input, and output a driving angle. Our white-box detector can identify weak points with f1 score up to 78.9%.

In summary, we make the following contributions:

- We conduct an empirical study to understand the local robustness properties of DNNs under natural variations.
- We develop a white-box (DEEPROBUST-W) and a black-box (DEEPROBUST-B) method to automatically detect weak points.
- We present a detailed evaluation of our methods on three DNN models across three image classification datasets. To check the generalizability of our findings, we further evaluate DEEPROBUST-W in a setting with non-spatial transformations (*i.e.* rain and fog), a different task (*i.e.* regression), and a safety-critical application (*i.e.* self-driving car). We find that DEEPROBUST can successfully detect weak points with reasonably good precision and recall.
- We made our code public at `https://github.com/deeprobust/DeepRobust`.

4.2 Methodology

4.2.1 Terminology

Original Data Point: An original data point represents an original un-modified data instance (image in our case) in the studied dataset. The original data points can come from training, validation, or testing dataset, depending on the experimental setting. In Figure 4.2, the triangle in the center is an original data point.

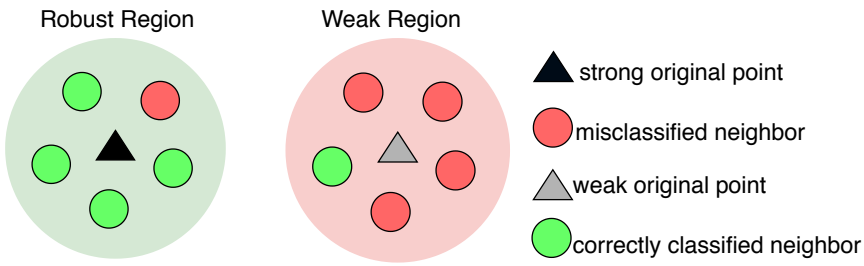


Figure 4.2: **Illustrating our terminologies. The triangles are original points, and the small circles are their neighbors generated by natural variations. The light-green region is robust with higher neighbor accuracy, while the light-red region is vulnerable. The corresponding original points are robust and non-robust accordingly.**

Neighbors: Neighbors are images generated by the natural variations, *e.g.,* spatial transformations applied to an original image. Since the transformation parameters are continuous (e.g., degree of rotations), there can be an infinite number of neighbors per image. In Figure 4.2, the small circles around an original data point represent its neighbors.

Neighbor Accuracy: We define *neighbor accuracy* as the percentage of its neighbors, including itself, that can be correctly classified by the DNN model. Figure 4.2 illustrates this; here, red small circles indicate misclassified neighbors, while the green small circles are correctly classified ones. The figure shows that there are only five neighbors per original data point. In the left-hand-side diagram, four out of five neighbors are correctly classified by the given DNN model. If the original data point is correctly classified as well, the neighbor accuracy of the original data is (5/6=) 83.3%. Similarly, in Figure 4.2 (right), four out of the five neighbors have been misclassified by the model; if the original data point is misclassified, the neighbor accuracy is (1/6=) 16.6%.

Robustness. An original data point is strong, *a.k.a.* robust, w.r.t. the DNN model under test if its neighbor accuracy is higher than a pre-defined threshold. Conversely, a weak, *a.k.a.* non-robust, point has the neighbor accuracy lower than a pre-defined threshold. For example, at 0.75 neighbor accuracy threshold, the black triangle in Figure 4.2 is a strong point, and the grey triangle is a weak point.

A region contains an original point and all of its neighbors. If the original point is strong (weak), we call the corresponding region as a robust (weak) region. In Figure 4.2, the light green

region is robust while the light red region is weak.

Neighbor Diversity: For multi-class classification task, different neighbors of an original point can be mis-classified to different classes. Neighbor Diversity score measures how many diverse classes a point's neighbors are classified, and is formally computed using Simpson Diversity Index (λ) [202]:

$$\lambda = \sum_{i=1}^{k} p_i^2 \tag{4.1}$$

where k is the total number of possible classes and p_i is the probability of an image's neighbors being predicted to be class i. Large Simpson Index means low diversity. Let's consider we have three possible classes A, B, and C. Assume an image has 4 neighbors. Including the original image, there are 5 images in total. If two of the five images are classified as A, and rest are classified as B, then $\lambda = (2/5)^2 + (3/5)^2 + (0/5)^2 = 0.52$. In contrast, if two of them are classified as A, and two are classified as B, and one is classified as C then $\lambda = (2/5)^2 + (2/5)^2 + (1/5)^2 = 0.36$. Clearly, the latter case is more diverse and thus, has a lower λ score.

Feature Representation: In a DNN, the neurons' output in each layer capture different abstract representation of the raw input, which are commonly known as features, extracted by the current layer and all the preceding layers. Each layer's output forms the corresponding feature space. For a given input data point, we consider the output of the DNN's second-to-last layer as its feature representation or feature vector.

4.2.2 Data Collection

Neighbor Generation: For the image classification tasks, for each original image point, we generate its neighbors by combining two types of spatial transformations: rotation and translation. We carefully choose these two types as representatives of non-linear and linear spatial transformations, respectively, following Engstrom et al. [47]. In particular, following them, we generate a neighbor by randomly rotating the original point by t ($\in [-30, 30]$) degrees, shifting it by dx (about 10% of the original image's width i.e. $\in [-3, 3]$) pixels horizontally, and shifting it by dy (about 10% of the original image's height i.e. $\in [-3, 3]$) pixels vertically. It should be noted that for image

classification it is standard in the literatures [71, 47, 203] to assume that the transformed image has the same label as the original one. As the transformation parameters are continuous, there can be infinite neighbors of an original data point. Hence, we sample m neighbors for each original data point. We explore the impact of m in RQ2.

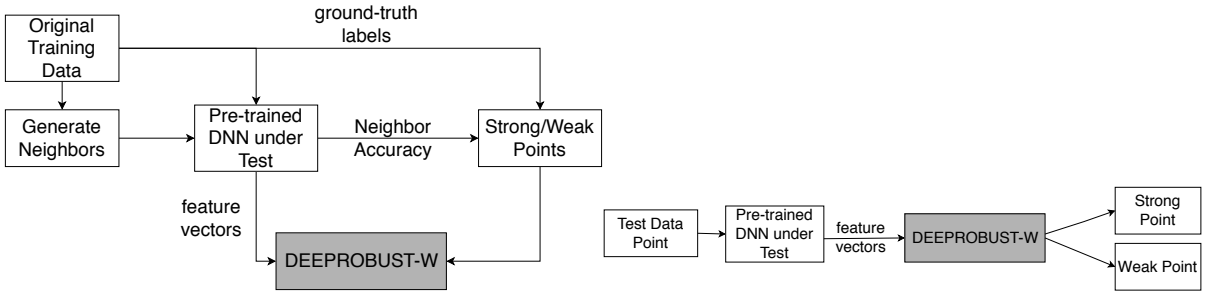
For the self-driving-car task where the model predicts steering angle, for each original image point, we generate 50% neighbors with rain effect and the rest 50% with fog effects. We adopt a widely used self-driving car data augmentation package, Automold [204], for adding these effects where we randomly vary the degrees of the added effect. For the rain effect, we set "rain_type=heavy" and everything else as default. For the fog effect, we set everything as default.

Estimating Neighbor Accuracy: To compute the neighbor accuracy of a data point for a given DNN model, we first generate its neighbor samples by applying different transformations—spatial for image classification and rain or fog for self-driving-car application. Then we feed these generated neighbors into the DNN model and compute the accuracy by comparing the DNN's output with the label of the original data point. For self-driving-car application, we follow the technique described in DeepTest [45]. More specifically, if the predicted steering angle of the transformed image is within a threshold to the original image, we consider it as correct. This ensures that any small variations of steering angle are tolerated in the predicted results. We then compute $neighbour\ accuracy = \frac{\#correct\ predictions}{original\ point+\#total\ neighbours}$.

4.2.3 Classifying Robust vs. Weak Points

We propose two methods, DEEPROBUST-W and DEEPROBUST-B, to identify whether an unlabeled input is strong or weak *w.r.t.* a DNN in real time. If a test image is identified as a weak point, although it may be classified correctly by the pre-trained model, this image is in a vulnerable region where a slight change to this image may cause the pre-trained DNN to misclassify the changed input.

DEEPROBUST-W: White-box Classifier



(a) workflow - training (b) workflow - testing

Figure 4.3: **Workflow of DEEPROBUST-W**

This is a binary classifier designed to classify an image (in particular, image feature vector) as a strong or weak point. Here, we assume that we have white box access to the DNN under test to extract the feature vectors of the input images from the DNN. These feature vectors are given as inputs to DEEPROBUST-W. Figure 4.3 shows the workflow.

Training: During training of DEEPROBUST-W, we first feed all the original training images and their neighbors to the DNN under test. From the DNN outputs, we compute the neighbor accuracy for each data point in the training set and label each point strong/weak depending on whether its neighbor accuracy is higher/lower than a predefined threshold. For each original data point, we also extract the output of the DNN's second-to-last layer as its feature vector. We use these vectors as inputs to train DEEPROBUST-W and the outputs are the corresponding strong/weak labels.

Testing: Given a test input, we first extract its feature vector by feeding the test image to the DNN under test and then feed the extracted feature vector into trained DEEPROBUST-W, which predicts if the input image is a strong or weak point.

DEEPROBUST-B: Black-box Classifier

This is also a binary classifier that is intended to classify an image to strong/weak point. However, here the user does not have white box access to the DNN under test. Figure 4.4 shows the workflow.

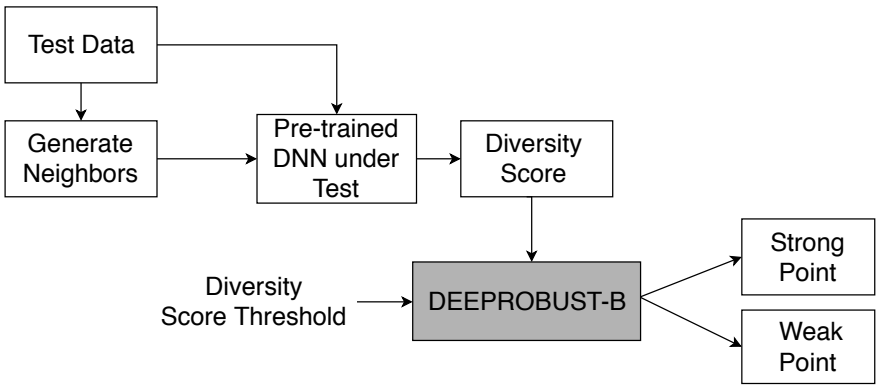


Figure 4.4: **Workflow of DEEPROBUST-B**

Given a test input, we first randomly generate some of its neighbors. We then query the DNN under test with all these neighbors and compute the diversity score, as per Equation 4.1. If the neighbor diversity score (inversely correlated with neighbor diversity) is greater than a given diversity score threshold, the given test input is classified as a strong point; otherwise, a weak point.

Notice that, in this method, we do not need a training step. We only need the diversity score threshold, which can be empirically set using a ground-truth data set. In particular, we first calculate the neighbor accuracy and diversity score of each pre-annotated point. Next, based on a given neighbor accuracy threshold, we identify the weak points, as the ground truth. The highest diversity score among these weak points is chosen as the diversity score threshold.

Usage Scenario

DEEPROBUST-W/B works in a real-world setting where a customer/user runs a pre-trained DNN model in real-time which constantly receives inputs and wants to test if the prediction of the DNN on a given input can be trusted. DEEPROBUST-W assumes that the user has white-box access to DNN under test and all the training data used to train the DNN. DEEPROBUST-W leverages the feature vector and neighbor accuracy of the training data to train the classifier, which can notify the user if the current input is a strong point or weak point. If the input is classified as strong point, the user can give more trust to the original DNN's prediction. On the other hand, if the point is classified as a weak point, the user may want to be more cautious about the DNN's prediction and

conduct additional inspections.

In the blackbox setting, DEEPROBUST-B assumes the user does not have white-box access to DNN under test. DEEPROBUST-B comes with a small overhead of transforming the input multiple times to get some neighbors and querying DNN under test on them to estimate the diversity score.

4.3 Experimental Design

4.3.1 Study Subjects

Image Classification

Similar to many existing works [45, 46, 196, 50, 205, 78] on DNN testing, in this work, we use image classification application of DNNs as the basis of our investigation. This is one of the most popular computer vision tasks, where the model tries to classify the objects in an image or video. **Datasets:** We conduct our experiments on three image classification datasets: F-MNIST [206], CIFAR-100 [207], and SVHN [208].

- **CIFAR-10**: consists of 50,000 training and 10,000 testing 32x32 color images. Each image is one of ten digit classes.
- **F-MNIST**: consists of 60,000 training images and 10,000 testing 28x28 grayscale images. Each image is one of ten fashion product related classes.
- **SVHN**: consists of 73,257 training images and 26,032 testing images. Each image is a 32x32 color cropped image of house numbers collected from Google Street View images.

Architectures: The popular DNN-based image classifiers are variants of convolutional neural networks (CNN) [209, 210, 92]. Here we study the following three architectures for all the three datasets:

- **ResN**: Following Engstrom *et al.* [47], we use ResN model with 4 groups of residual layers with filter sizes 16, 16, 32, and 64, and 5 residual units each.

Table 4.1: **Study Subjects (values are in percentage)**

Dataset	CIFAR-100			SVHN			F-MNIST		
Model	VGG	ResN	WRN	VGG	ResN	WRN	VGG	ResN	WRN
nat acc'	89.0	89.3	90.6	94.5	95.3	95.2	93.4	93.5	93.6
rob acc*	75.5	68.5	74.8	78.1	78.9	81.	61.1	63.0	64.2

'Natural accuracy. *Robust accuracy is estimated as the average neighbor accuracy for test data points.

- **VGG**: We use the same VGG architecture as proposed in [211].
- **WRN**: We use a structure with block type (3, 3) and depth 28 in [1] but replace the widening factor 10 with 2 for less parameters and faster training.

We train all the models from scratch. When training models on CIFAR-100 , we pre-process the input images with random augmentation (random translation with $dx, dy \in [-2, 2]$ pixels both horizontally and vertically) which is a widely used preprocessing step for this dataset. When training models on the other two datasets, plain images are directly fed into the models. The natural accuracies and robust accuracies of the models are shown in Table 4.1.

Steering Angle Prediction

We further evaluate DEEPROBUST-W in a self-driving car application to show that it can be applied into a regression task. These models learn to steer (*i.e.* predict steering angle) by taking in visual inputs from car-mounted cameras that record the driving scene, paired with the steering angles from a human driver.

Datasets: We use the dataset by Stocco *et al.* [212], which is collected by the authors driving on three tracks of different environments in the Udacity Simulator [213]. It consists of 37888 central camera training images and 9427 central camera evaluation images. Each image is of size 320x120.

Architectures: We evaluate our method on the three pre-trained DNN models used in [212]: NVIDIA DAVE-2 [214], Epoch [168], and Chauffeur [167]. These models have been used by many previous testing works on self-driving car [44, 45, 212].

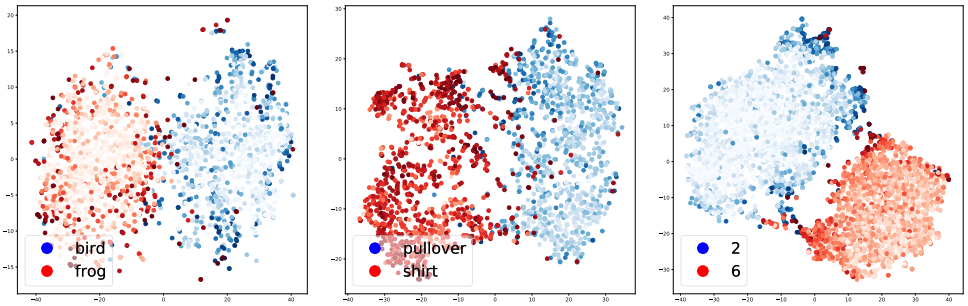
4.3.2 Evaluation

Evaluation Metric. We evaluate both DEEPROBUST-W and DEEPROBUST-B for detecting weak points under twelve and nine different DNN-dataset combinations, respectively, in terms of precision, recall, and F1 score. Let us assume that E is the number of weak points detected by our tool and A is the the number of true weak points in the ground truth set. Then the precision and recall are $\frac{|A \cap E|}{|E|}$ and $\frac{|A \cap E|}{|A|}$, respectively. F1 score is a single accuracy measure that considers both precision and recall, and defined as $\frac{2 \times precision \times recall}{precision + recall}$. We perform each experiment for two thresholds of neighbor accuracy that defines strong vs. weak points: 0.75 and 0.50.

Baselines. We compare DEEPROBUST-W and DEEPROBUST-B with two baselines. One naive baseline (denoted *random*) is randomly selecting the same number of points as detected by our proposed method to be weak points. Another baseline (denoted *top1*) is based on prediction confidence score—if the confidence of a data point is higher than a pre-defined cutoff we call it a strong point, weak otherwise. This baseline is based on the intuition that DNNs might not be confident enough to predict the weak points.

4.4 Results

In this section, we elaborate on our results. In our preliminary experiments, we have two findings regarding neighbor accuracy. First, the neighbor accuracy vary widely across data points and there is a non-trivial number of points having relatively low neighbor accuracy. For example, for all the models trained on CIFAR-100 dataset, 40% of training data and 42% of testing data have neighbor accuracy <0.75, and 16% of training data and 20% of testing data have neighbor accuracy <0.50. These points degrade the aggregated spatial robustness of the model. The same finding holds for the other two datasets. Second, the distribution of neighbor accuracy for a dataset is similar across different models. For CIFAR-100 , F-MNIST and SVHN, 60%, 76%, and 81%, respectively, of data points have neighbor accuracy change < 0.2 across any two models on the same dataset. This implies that a large portion of data points' neighbor accuracy is independent of



(a) CIFAR-100 (b) F-MNIST (c) SVHN

Figure 4.5: **The t-SNE plots of data points from two randomly chosen classes across all three datasets when using ResNet. Darker color indicates lower neighbor accuracy.**

the model selected.

The first observation shows that neighbor accuracy is a distinguishable measure for local robustness for the datasets and models we study. The second observation implies that the properties of points of low neighbor accuracy may be similar across models for each dataset. Following these two observations, we dive deeper and explore the characteristics of data points with different neighbor accuracy in RQ1. We then evaluate the performance of DEEPROBUST-W and DEEPROBUST-B which are developed based on the observations from RQ1 in RQ2 and RQ3, respectively. Finally, in RQ4, we evaluate the generalizability of our method by applying DEEPROBUST-W in a regression task for self-driving cars under more complex transformations.

RQ1. What are the characteristics of the weak points? Here we explore the characteristics of robust vs. non-robust points in their feature space. In particular, we check the difference in feature representations between: a) robust and non-robust points, and b) points with different degrees of robustness.

RQ1a. Given a well trained model, do the feature representations of robust and non-robust points vary? In this RQ, we first explore how robust (*i.e.* strong) and non-robust (*i.e.* weak) data points are distributed in the feature space. We apply t-SNE[215], a widely used visualization method, to visualize the distribution of points of different neighbor accuracy in the representation space for all three datasets when using ResN as the classifier. Figure 4.5 shows the visualization of feature vectors from two randomly picked classes with colors indicating the neighbor accuracy of each

Table 4.2: **Weak and strong points ratio, and cohen's d effect size**

Dataset	CIFAR-100			SVHN			F-MNIST		
Model	ResN	WRN	VGG	ResN	WRN	VGG	ResN	WRN	VGG
Neighbor Accuracy Cutoff=0.5									
r_w	0.915	0.955	1.004	1.046	1.103	0.997	0.746	0.734	0.976
r_s	0.609	0.584	0.975	0.294	0.309	0.977	0.297	0.293	0.930
d*	1.368	1.736	1.163	2.077	2.428	1.420	1.426	1.312	1.332
Neighbor Accuracy Cutoff=0.75									
r_w	0.778	0.796	0.992	0.604	0.671	0.983	0.516	0.496	0.953
r_s	0.588	0.558	0.973	0.260	0.274	0.977	0.253	0.257	0.918
d*	0.786	1.040	0.749	0.860	1.111	0.401	0.749	0.642	0.937

*Cohen's d effect size of 0.20 = small, 0.50 = medium, 0.80 = large, 1.20 = very large, and 2.0 = huge [216, 217].

point. The darker a point's color is, the lower its neighbor accuracy is. It is evident that most points of low neighbor accuracy tend to be further away from the class center.

To numerically verify this observation, first, we define a class center c_k for each class k as the median value of the feature vectors of all the points from class k. Thus, if f_i is the feature of a point at ith dimension and $\hat{f_{ik}}$ is the median of the ith dimension features for all the points in class k, c_k is defined to be $(\hat{f_{1k}}, ..., \hat{f_{jk}}, ..., \hat{f_{nk}})$.

The reason we take median rather than mean is that it is a more statistically stable measure and is less likely to be heavily influenced by outliers in the representation space. Then, for every point p, we define a ratio: $r^{(p)} = \frac{d^{(p)}_{same\_class}}{d^{(p)}_{nearest\_other\_class}}$, where $d^{(p)}_{same\_class}$ is the distance of the p-th point's feature vector to its own class center and $d^{(p)}_{nearest\_other\_class}$ is the distance of the p-th point's feature vector to the class center of its closest other class. A small $r^{(p)}$ means that the point p is close to its own class center while far from other classes, *i.e.* p is far from the decision boundary. In contrast, a larger $r^{(p)}$ indicates that the point p is closer to some other classes, *i.e.* it is closer to the decision boundary.

We then measure the average $r^{(p)}$ among the weak points (denoted as r_w) and among strong points (denoted as r_s) for all three datasets across three models. Besides, we also calculate mann-whitney wilocox test[218] and cohen's d effect size [216] between the two ratios to test if the two ratios indeed have statistically significant difference and how large the difference is.

Table 4.3: **Spearman Correlation between Neighbor Accuracy and Simpson Diversity Index. All coefficients are reported with statistical significance ($p < 0.05$).**

Dataset	CIFAR-100			SVHN			F-MNIST		
Model	ResN	WRN	VGG	ResN	WRN	VGG	ResN	WRN	VGG
corr.coeff.	0.853	0.909	0.946	0.970	0.984	0.983	0.923	0.962	0.8947

As shown in Table 4.2, for both the neighbor accuracy cutoff (0.5 and 0.75), except one setting, the cohen's d effect size for every setting is larger than 0.50, which implies a medium to very large difference. Besides, for every setting, the mann-whitney wilocox test value (not shown in the table) is smaller than $1e^{-80}$, which implies the difference is indeed statistically significant.

The visualization and numerical results imply that most weak points are close to the decision boundaries between classes. Note that similar observation was also observed by Kim et. al. [196] in case of adversarial perturbation. In particular, they find that adversarial examples tend to be closer to class decision boundaries. In contrast, we focus on spatial robustness and find that spatially non-robust points are closer to decision boundaries.

RQ1b. Given a well trained model, do the feature representations of the data points vary by their degree of robustness? By analyzing the classifications of the neighbors of weak vs. strong points, we observe that the weaker a point is, its neighbors are more likely to be classified in different classes. We quantify this observation by computing diversity of the outputs a point's neighbor; We adopt Simpson Diversity Index (λ) [202] as defined in Equation (4.1).

Table 4.3 shows the Spearman correlation between neighbor accuracy and λ on the three datasets and three models for each. Note that while calculating the correlation, we remove points with neighbor accuracy 100% since there are many points having 100% neighbor accuracy and tend to bias upward the Spearman Correlation; if we include points with neighbor accuracy 100%, the correlations become even higher. We notice that for any setting, the Spearman Correlation is never lower than 0.853. This indicates that neighbor accuracy and diversity are highly correlated with each other. For example, the bird image in Fig.4.1a has neighbor accuracy 0.49 and diversity 0.36, while the bird image in Fig.4.1e has neighbor accuracy 1 and diversity 1. This shows, the

classifier tends to be confused about weak points and mispredicts them into many different kinds of classes.

Result 1: *In the representation space, weak points tend to lie towards the class decision boundary while the strong points lie towards the center. The weaker an image is, the model tends to be more confused by it, and classify its neighbors into more diverse classes.*

RQ2. Can we detect the weak points in a white-box setting?

We explore this RQ using DEEPROBUST-W, as discussed in Section 4.2.3. DEEPROBUST-W takes the feature vector of a data point as input and classifies it to a strong/weak point. We implement DEEPROBUST-W with a simple 4-layer, fully connected neural network architecture with hidden layer dimensions 1500, 1000, and 500, respectively.

Table 4.4 shows the result. At 0.75 setting, DEEPROBUST-W has F1 up to 91.4%, with an average of 76.9%. At 0.50 setting, DEEPROBUST-W detects weak points with average f1 of 61.1%, while it can go as high as 79.1%. DEEPROBUST-W consistently performs significantly better than the top1 baseline and random baseline.

The top1 has very good precision, since a mis-classified image with low confidence tends to have very poor local robustness. However, there also exist many images that are correctly classified with high confidence yet have poor local robustness. The miss of these points leads the top1 to have very poor recall and thus even worse f1 compared with the random baseline. Our method comes to aid by providing high recall at the same time of decent precision.

Notice that DEEPROBUST-W's performance depends on the training data selection, mainly (a) how many weak vs. strong points are used to train the model, and (b) how many neighbors are generated per point to decide whether it is strong/weak. To investigate the previous one, we assign a weight to each input point, indicating how likely it would be selected to train DEEPROBUST-W. In particular, for an input i, a weight $w_i := \frac{1+(1-n_i)^m \times 100^m}{1+100^m}$ is computed, where n is its neighbor accuracy, and m is a configurable parameter; with larger m, more weak points are sampled, and

Table 4.4: **Performance of DEEPROBUST-W and the baseline methods for predicting weak points.**

dataset	model	method	0.75 neighbor acc.			0.50 neighbor acc.		
			f1	tp	fp	f1	tp	fp
CIFAR-100	ResN	ours	0.79	3844	764	0.581	1290	664
		top1	0.376	1218	206	0.182	255	120
		random	0.488	2372	2236	0.233	520	1445
	WRN	ours	0.747	2901	906	0.56	947	610
		top1	0.35	889	222	0.183	189	90
		random	0.395	1534	2273	0.154	261	1296
	VGG	ours	0.654	2222	938	0.493	747	543
		top1	0.439	1070	153	0.266	278	106
		random	0.332	1127	2033	0.132	200	1090
SVHN	ResN	ours	0.755	6814	2530	0.577	1414	674
		top1	0.315	1665	142	0.267	452	122
		random	0.343	3095	6249	0.086	210	1878
	WRN	ours	0.709	5062	2143	0.582	1404	1055
		top1	0.292	1238	130	0.203	275	85
		random	0.28	2000	5205	0.095	229	2230
	VGG	ours	0.595	5214	3367	0.498	1272	911
		top1	0.172	840	67	0.139	221	52
		random	0.341	2986	5595	0.094	240	1943
F-MNIST	ResN	ours	0.914	6034	873	0.791	2144	556
		top1	0.124	428	11	0.039	57	7
		random	0.657	4340	2567	0.263	712	1988
	WRN	ours	0.896	5743	652	0.76	2033	641
		top1	0.144	490	14	0.045	63	8
		random	0.638	4093	2302	0.281	752	1922
	VGG	ours	0.864	6348	1231	0.654	1895	1082
		top1	0.104	392	5	0.028	39	5
		random	0.734	5393	2186	0.295	854	2123

vice versa. Thus, if m is larger, DEEPROBUST-W will be trained with more weak points and vice versa.

Table 4.5A shows the performance: as m increases, the detector trades precision for recall. In this way, choosing different values of m, the precision-recall trade-off of the detector can be adjusted according to a user's need. From a different perspective, this way of oversampling weak points also addresses the potential problem of imbalanced data when the weak points are much less than the strong points.

Table 4.5: **DEEPROBUST-W performance using different sampling strategies for training**

A: with varying number of strong/weak points

dataset	m	prec	recall	tp	fp	f1
CIFAR-100	0	**0.660**	0.518	1290	664	0.581
	1	0.615	0.599	1490	932	0.607
	2	0.544	**0.699**	1740	1460	**0.612**
SVHN	0	**0.677**	0.502	1414	674	0.577
	1	0.575	0.653	1837	1357	**0.612**
	2	0.332	**0.767**	2160	4356	0.463
F-MNIST	0	**0.794**	0.787	2144	556	**0.791**
	1	0.746	0.839	2284	777	0.79
	2	0.712	**0.871**	2372	962	0.783

B: with varying number of neigbours

dataset	#neighbors	prec	recall	tp	fp	f1
CIFAR-100	6	0.662	0.389	967	493	0.49
	12	0.685	0.384	955	440	0.492
	25	0.665	0.502	1250	629	0.572
	50	0.660	0.518	1290	664	0.581
	200	0.683	0.507	1261	585	0.582
SVHN	6	0.723	0.403	1136	436	0.518
	12	0.672	0.527	1483	725	0.59
	25	0.619	0.629	1771	1090	0.624
	50	0.632	0.605	1703	993	0.618
	200	0.667	0.550	1550	774	0.603
F-MNIST	6	0.817	0.727	1981	443	0.77
	12	0.784	0.790	2153	592	0.787
	25	0.773	0.787	2143	629	0.78
	50	0.836	0.727	1981	390	0.778
	200	0.778	0.812	2211	632	0.794

Next, we check how DEEPROBUST-W's performance is dependent on the number of sampled neighbors, because a data point can potentially have infinite neighbors. Table 4.5B shows that the number of neighbors does not have much influence on the performance of the detector once it goes beyond some value (F1 score does not change more than 3.5 percentage point between 25 and 200 samples) for all the three datasets. Thus, we choose 50 for all of our experiments

Result 2: DEEPROBUST-W *can identify weak points with reasonably high F1 score: on average* 76.9%*, at* 0.75 *neighbor accuracy cut-off.*

RQ3. Can we identify the weak points in a black-box setting?

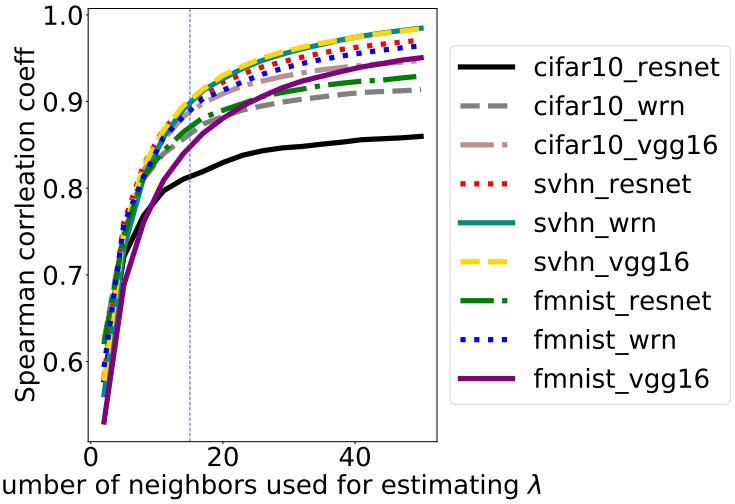


Figure 4.6: **The spearman correlation coeff. between diversity score (λ) and neighbor accuracy, with varying #neighbors (m).**

We explore this RQ using DEEPROBUST-B, as discussed in Section 4.2.3. Here, we assume, we only have access to unlabeled testing data and black-box access of the model under test. To evaluate DEEPROBUST-B, we spatially transform each test input m times by randomly applying $d\omega \in [-30, +30]$ degrees rotation, $dx \in [-3, +3]$ pixels horizontal translation, and $dy \in [-3, +3]$ pixels vertical translation. We then calculate the output diversity score (λ) based on Equation (4.1) and rank the test images based on λ. Finally, we mark top k images as potential most non-robust points. The parameter k is chosen according to users' need.

With each test data, DEEPROBUST-B queries the model with m neighbors to compute λ. Since querying the classifier comes with an overhead, our goal is to achieve an optimal accuracy with minimal queries (*i.e.* m). To determine an optimal m value, we explore the spearman correlation between diversity score and neighbor accuracy, with varying m, when running ResN on all the three datasets (see Figure 4.6). The correlation increases as m increases, as with more query λ becomes more accurate, and so the neighbor accuracy. We notice that at $m = 15$, the correlation coefficients across all the experimental settings reach above 0.8, and the rate of increase begins to slow down significantly. The results for the other two architectures are highly similar. Thus, we set $m = 15$ as default for DEEPROBUST-B.

Next, we evaluate DEEPROBUST-B's performance. We plot AUC-ROC by changing $top - k$

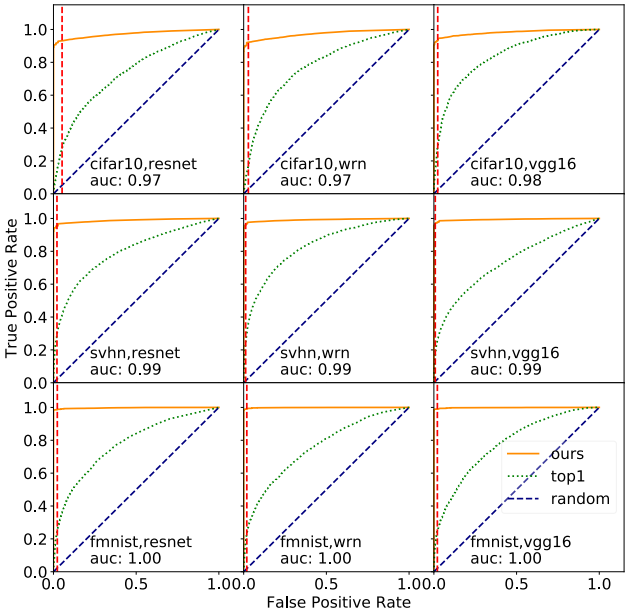


Figure 4.7: **AUC-ROC curve with neighbor accuracy cutoff at 0.75. The red vertical line indicates when the diversity score threshold is chosen from training data.**

at $m = 15$ and compare our method with the random baseline and the top1 baseline as before. As shown in Figure 4.7, our method performs much better than the random baseline. In particular, our proposed method achieves AUC higher than 0.87 for all settings when neighbor accuracy cutoff is 0.5 and 0.97 when neighbor accuracy cutoff is 0.75.

Instead of above ranking based scheme, DEEPROBUST-B can also be used as a classifier if a diversity threshold is given (see Section 4.2.3). Here, we estimate the threshold using pre-annotated training data.

We evaluate precision and recall of DEEPROBUST-B in the nine DNN-dataset combinations under neighbor accuracy cutoffs 0.5 and 0.75. Table 4.6 shows the result. At 0.75 setting, DEEPROBUST-B has f1 up to 99.1%, with an average of 96.5%. At 0.50 setting, DEEPROBUST-B detects weak points with average f1 of 72.9%, while it can go as high as 85.7%. It consistently produces much better estimation than the top1 baseline and the random baseline. This shows that our black-box method can effectively identify weak points.

Note that, generating the spatial transformations and querying the model with it under black box setting is fast. Previous black box methods for adversarial perturbation work in such fashion

Table 4.6: **Performance of DEEPROBUST-B and the random baseline method for predicting neighbor accuracy under different settings.**

dataset	model	method	75%			50%		
			f1	tp	fp	f1	tp	fp
CIFAR-100	ResN	ours	0.939	4714	257	0.622	1454	801
		top1	0.376	1218	206	0.182	255	120
		random	0.501	2516	2455	0.234	549	1706
	WRN	ours	0.938	3657	171	0.585	986	604
		top1	0.35	889	222	0.183	189	90
		random	0.383	1494	2334	0.182	307	1283
	VGG	ours	0.945	3397	148	0.682	1087	390
		top1	0.439	1070	153	0.266	278	106
		random	0.36	1296	2249	0.153	244	1233
SVHN	ResN	ours	0.956	8371	365	0.67	1845	858
		top1	0.315	1665	142	0.267	452	122
		random	0.336	2944	5792	0.102	280	2423
	WRN	ours	0.963	6827	227	0.718	1602	514
		top1	0.292	1238	130	0.203	275	85
		random	0.275	1950	5104	0.085	191	1925
	VGG	ours	0.976	8608	144	0.779	2138	454
		top1	0.172	840	67	0.139	221	52
		random	0.339	2997	5755	0.102	279	2313
F-MNIST	ResN	ours	0.987	6422	81	0.802	2316	546
		top1	0.124	428	11	0.039	57	7
		random	0.655	4265	2238	0.289	835	2027
	WRN	ours	0.989	6246	70	0.857	2297	360
		top1	0.144	490	14	0.045	63	8
		random	0.631	3987	2329	0.274	736	1921
	VGG	ours	0.991	7078	60	0.847	2393	418
		top1	0.104	392	5	0.028	39	5
		random	0.711	5084	2054	0.277	784	2027

[219, 220]. For example, using CIFAR-100 , when we use a batch with size 100, the average transformation+query time for one image is 0.031 ± 0.015 ms. For the other two datasets, the overhead is similar. Thus, to for $m = 15$ queries, it takes only 0.465 ± 0.225 ms, which is a negligible overhead for most real-world DNN based vision applications. This implies that our black-box method can also be used in real time for many applications.

Result 3: *Given only black-box access to the DNN classifier,* DEEPROBUST-B *can identify weak points with f1 that are much better than those of using top1 method or random method.*

RQ4. How generalizable are these findings?

The local robustness issues also exist in more critical applications like self-driving-car. Here we explore more complex transformations, *i.e.* adding rain and fog to the driving scenes. As shown in Figure 4.8, among those correctly classified data points, there is a non-trivial portion (45.8%) of them (in the heatmap, more red signified weaker) suffer from low (<0.75) neighbor accuracy.

Note that, here, we test regression models, which take images of driving scenes as inputs and output the corresponding steering angles.

Let the set of outputs predicted by a DNN be denoted by $\{\hat{\theta}_{o1}, \hat{\theta}_{o2}, ..., \hat{\theta}_{on}\}$, and corresponding ground truth labels for the original (unmodified) image points be $\{\theta_1, \theta_2, ..., \theta_n\}$. If the difference between predicted steering angle $\hat{\theta}_{oi}$ of a transformed image and the ground truth label of the original image θ_i is above a threshold,

we consider it as incorrect. The threshold λMSE_{orig} is defined following DeepTest's [45], where $MSE_{orig} = \frac{1}{n}\sum_{i=1}^{n}(\theta_i - \hat{\theta}_{oi})^2$. MSE is the Mean Square Error between the outputs and the manual labels, and λ is a positive coefficient that is chosen to reflect a user's tolerance on the deviation. Note that there is no softmax layer (and thus no confidence score) in these regression models so the top1 baseline method cannot be used here.

Table 4.7 shows the result when $\lambda = 3$. At 0.75 setting, DEEPROBUST-W has f1 score up to 78.9%, with an average of 58.2%. At 0.50 setting, DEEPROBUST-W detects weak points with an average f1 of 47.9%, while it can go as high as 68.2%. It consistently produces much better

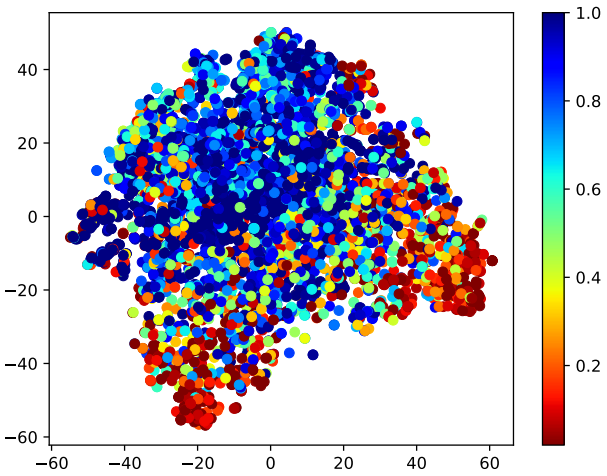


Figure 4.8: **The t-SNE plot of correctly classified data points from Self-Driving dataset by the epoch model. data points are colored based on neighbor accuracy.**

model	method	0.75 neighbor acc.			0.50 neighbor acc.		
		f1	tp	fp	f1	tp	fp
chauffeur	ours	0.417	555	547	0.346	339	384
	random	0.146	194	908	0.096	94	629
epoch	ours	0.789	4354	1112	0.682	2641	1127
	random	0.586	3234	2232	0.411	1592	2176
dave2	ours	0.541	979	471	0.409	475	246
	random	0.193	350	1100	0.121	141	580

Table 4.7: **Performance of DEEPROBUST-W for predicting weak points of Self-Driving dataset**

estimation than the random baseline under all the settings. It should be noted that our observation is valid for all the λ used in [45] from λ equal to 1 to 5. This shows that our proposed method DEEPROBUST-W can be applied to regression problems with more complex natural transformations.

It should also be noted that it is unrealistic to use DEEPROBUST-B for this task for two reasons: It is impractical to try different variations of an image in real-time for a self-driving car, which is a time-sensitive application. Further, DEEPROBUST-B requires the calculation of neighbor diversity score. For a regression problem, the predicted values are continuous, so there is a very low probability for any two predictions being equal. Thus, the neighbor diversity score for every data point will be the same and cannot be used for identifying the weak points.

Result 4: DEEPROBUST-W *can detect weak points of a self-driving car dataset with f1 score up to 78.9%, with an average of 58.2%, at neighbor accuracy cutoff 0.75.*

4.5 Discussion & Threats to Validity

We adopt rotation and translation as transformations for image classification tasks and rain and fog effects for the self-driving car task. There are many more natural variations such as brightness, snow effect etc. However, rotation and translation are representative of spatial transformation and used by many paper in evaluating robustness of DNN models[47, 44]. Rain and fog effects are also widely leveraged in many influential studies on testing self-driving cars [44, 45, 46].

Besides, for some of the experiments we did not show all the combinations under both neighbor accuracy cutoffs (i.e. 0.5 and 0.75). However, we note that the observations are consistent and we did not include them purely because of space limitation. Another limitation is that for both DEEPROBUST-W and DEEPROBUST-B, we need to decide the number of neighbors to use for training a classifier and estimating λ, respectively. We mitigate it by selecting the neighbor numbers that give stable performance in terms of precision and recall.

4.6 Related Work

Adversarial examples. Many works focus on generating adversarial examples to fool the DNNs and evaluate their robustness using pixel-based perturbation [113, 123, 124, 127, 221, 196, 70, 222, 66, 126, 130, 129, 223, 224]. Some other papers [203, 71, 47], like us, proposed more realistic transformations to generate adversarial examples. In particular, Engstrom et al. [47] proposed that a simple rotation and translation can fool a DNN based classifier, and spatial adversarial robustness is orthogonal to l_p-bounded adversarial robustness. However, all these works estimate the overall robustness of a DNN based on its aggregated behavior across many data points. In contrast, we analyze the robustness of individual data points under natural variations and propose methods to detect weak/strong points automatically.

DNN testing. Many researchers [44, 50, 57, 196, 78, 225, 226, 227, 228, 229] proposed techniques to test DNN. For example, Pei et al. [44] proposed an image transformation based differential testing framework, which can detect erroneous behavior by comparing the outputs of an input image across multiple DNNs. Ferit et al. [229] used fault localization methods to identify suspicious neurons and leveraged those to generate adversarial test cases.

In contrast, others [45, 46, 225, 228, 230, 231] used metamorphic testing where the assumption is the outputs of an original and its transformed image will be the same under natural transformations. Among them, some use a uncertainty measure to quantify some types of non-robustness of an input for prioritizing samples for testing / retraining [230] or generating test cases[231]. We follow a similar metamorphic property while estimating neighbor accuracy and our proposed DEEPROBUST-B also leverages an uncertainty measure. The key differences are: First, we focus on estimating model's performance on general natural variants of an input rather than the input itself or only spatial variants. Second, we focus on the task of weak points detection rather than prioritizing / generating test cases. We also give detailed analyses of the properties of natural variants and propose a feature vector based white-box detection method DEEPROBUST-W. Further, we show that our method works across domains (both image classification and self-driving

car controllers) and tasks (both classification and regression). Other uncertainty work complement ours in the sense that we can easily leverage weak points identified by DEEPROBUST-W and DEEPROBUST-B to prioritize test cases or generate more adversarial cases of natural variants.

Another line of work [232, 233, 234, 235, 236, 237, 238] estimates the confidence of a DNN's output. For example, [233] leverages thrown away information from existing models to measure confidence; [234] shows other NN properties like depth, width, weight decay, and batch normalization are important factors influencing prediction confidence. Although such methods can provide a confidence measure per input or its adversarial variants, they do not check its natural robustness property, i.e., with natural variations how will they behave.

DNN verification. There also exist work on verifying properties for a DNN model [239, 70, 156, 240, 241]. Most of them focus on verifying properties on l_p norm bounded input space. Recently, Balunovic et al.[49] provides the first verification technique for verifying a data point's robustness against spatial transformation. However, their technique suffers from scalability issues.

Robust training. Regular neural network training involves the optimization of the loss value for each data point. Robust training of neural network works on minimizing the largest loss within a specific bounded region usually using adversarial examples [66, 67, 68, 69, 70, 71, 72, 73, 74]. While both robust training methods and our work generate variants of data points, instead of training a model with these variants to improve robustness, we leverage them to estimate the robustness of the unseen data points. The relation between robust retraining and us is thus similar to bug fixing vs. bug detection in traditional software engineering literature.

4.7 Conclusion

In this work, we involve the data characteristic into the robustness testing of a DNN model. We adopt the concept of neighbor accuracy as a measure for local robustness of a data point on a given model. We explore the properties of neighbor accuracy and find that weak points are often located towards the corresponding class boundaries and their transformed versions are likely to be predicted to be more diverse classes. Leveraging these observations, we propose a white-

box method and a black-box method to identify weak/strong points to warn a user about potential weakness in the given trained model in real-time. We design, implement and evaluate our proposed framework, DEEPROBUST-W and DEEPROBUST-B, on three image recognition datasets and one self-driving car dataset (for DEEPROBUST-W only) with three models for each. The results show that they can effectively identify weak/strong points with high precision and recall.

For future work, other consistency analysis methods [238] e.g. variation ratio, entropy can be tried. We can also leverage ideas from [230, 231] to easily prioritize test cases or generate more hard test cases based on identified weak points. Further, we can potentially modify existing fixing methods such as [55] targeting the weak points to fix them.

Chapter 5: DeepInspect: Testing DNN Image Classifiers for Group-Level Errors

In Chapter 3, we introduced our work DeepTest, where we designed, implemented and evaluated our proposed techniques for systematic testing DNN based software. We show that our tool identified thousands of erroneous behaviours in three top performing DNN based self-driving car models in Udacity self-driving challenge. In Chapter 4, we introduced our work DeepRobust, where we studied the per-point robustness of DNNs under natural variation, proposed and implemented a white-box approach and black-box approach in identifying weak points for DNN based image classifiers and DNN based self-driving cars. These two works as well as most of existing works in SE for AI area focus on instance-wise errors, which are single inputs that result in a DNN model's erroneous outputs. In this chapter, we introduce our third project DeepInspect, where we first show another type of errors, group-level errors, which DNN based software also suffer from but most of existing works ignored. Then we categorized group-level errors into confusion errors and bias errors based on real-world reports. Lastly, we proposed neuron coverage based distance metric to automatically test DNN based software for group-level errors without requiring data labels.

We publicly release the source code[1]. All images, figures, tables, equations, and text included in this chapter is based on a published collaborative work[78].

5.1 Introduction

Image classification has a plethora of applications in software for safety-critical domains such as self-driving cars, medical diagnosis, *etc.* Even day-to-day consumer software includes image

[1] https://github.com/ARiSE-Lab/DeepInspect

classifiers, such as Google Photo search and Facebook image tagging. Image classification is a well-studied problem in computer vision, where a model is trained to classify an image into single or multiple predefined categories [242]. Deep Neural Networks (DNNs) have enabled major breakthroughs in image classification tasks over the past few years, sometimes even matching human-level accuracy under some conditions [209], which has led to their ubiquity in modern software.

However, in spite of such spectacular success, DNN-based image classification models, like traditional software, are known to have serious bugs. For example, Google faced backlash in 2015 due to a notorious error in its photo-tagging app, which tagged pictures of dark-skinned people as "gorillas" [79]. Analogous to traditional software bugs, the Software Engineering (SE) literature denotes these classification errors as *model bugs* [65], which can arise due to either imperfect model structure or inadequate training data.

At a high-level, these bugs can affect either an *individual image*, where a particular image is mis-classified (*e.g.,* a particular skier is mistaken as a part of a mountain), or an *image class*, where a class of images is more likely to be mis-classified (*e.g.,* dark-skinned people are more likely to be misclassified), as shown in Table 5.1. The latter bugs are specific to a whole *class* of images rather than individual images, implying systematic bugs rather than the DNN equivalent of off-by-one errors. While much effort from the SE literature on Neural Network testing has focused on identifying individual-level violations—using white-box [44, 50, 196, 221], grey-box [45, 65], or concolic testing [57], detection of class-level violations remains relatively less explored. This paper focuses on automatically detecting such class-level bugs, so they can be fixed.

After manual investigation of some public reports describing the class-level violations listed in Table 5.1, we determined two root causes: (i) **Confusion**: The model cannot differentiate one class from another. For example, Google Photos confuses skier and mountain [81]. (ii) **Bias**: The model shows disparate outcomes between two related groups. For example, Zhao *et al.* in their paper "Men also like shopping" [83], find classification bias in favor of women on activities like shopping, cooking, washing, *etc.* We further notice that some class-level properties are violated

Table 5.1: **Examples of real-world bugs reported in neural image classifiers**

Bug Type	Name	Report Date	Outcome
Confusion	Gorilla Tag [79]	Jul 1, 2015	Black people were tagged as gorillas by Google photo app.
	Elephant is detected in a room [80]	Aug 9, 2018	Image Transplantation (replacing a sub-region of an image by another image containing a trained object) leads to mis-classification.
	Google Photo [81]	Dec 10, 2018	Google Photo confuses skier and mountain.
Bias	Nikon Camera [82]	Jan 22, 2010	Camera shows bias toward Caucasian faces when detecting people's blinks.
	Men Like Shopping [83]	July 29, 2017	Multi-label object classification models show bias towards women on activities like shopping, cooking, washing, *etc.*
	Gender Shades[84]	2018	Open-source face recognition services provided by IBM, Microsoft, and Face++ have higher error rates on darker-skin females for gender classification.

in both kinds of cases. For example, in the case of *confusion errors*, the classification error-rate between the objects of two classes, say, skier and mountain, is significantly higher than the overall classification error rate of the model. Similarly, in the bias scenario reported by Zhao *et al.*, a DNN model should not have different error rates while classifying the gender of a person in the shopping category. Unlike individual image properties, this is a class property affecting all the shopping images with men or women. Any violation of such a property by definition affects the whole class although not necessarily every image in that class, *e.g.,* a man is more prone to be predicted as a woman when he is shopping, even though some individual images of a man shopping may still be predicted correctly. Thus, we need a class-level approach to testing image classifier software for confusion and bias errors.

The bugs in a DNN model occur due to sub-optimal interactions between the model structure and the training data [65]. To capture such interactions, the literature has proposed various metrics primarily based on either neuron activations [44, 50, 196] or feature vectors [65, 243]. However, these techniques are primarily targeted at the individual image level. To detect class-level violations, we abstract away such model-data interactions at the class level and analyze the inter-class interactions using that new abstraction. To this end, we propose a metric using neuron activations and a baseline metric using weight vectors of the feature embedding to capture the class abstraction.

For a set of test input images, we compute the probability of activation of a neuron per predicted class. Thus, for each class, we create a vector of neuron activations where each vector element

corresponds to a neuron activation probability. If the distance between the two vectors for two different classes is too close, compared to other class-vector pairs, that means the DNN under test may not effectively distinguish between those two classes. Motivated by MODE's technique [65], we further create a baseline where each class is represented by the corresponding weight vector of the last linear layer of the model under test.

We evaluate our methodology for both single- and multi-label classification models in eight different settings. Our experiments demonstrate that DeepInspect can efficiently detect both Bias and Confusion errors in popular neural image classifiers. We further check whether DeepInspect can detect such classification errors in state-of-the-art models designed to be robust against norm-bounded adversarial attacks [244]; DeepInspect finds hundreds of errors proving the need for orthogonal testing strategies to detect such class-level mispredictions. Unlike some other DNN testing techniques [45, 44, 243, 57], DeepInspect does not need to generate additional transformed (synthetic) images to find these errors. The primary contributions of this paper are:

- We propose a novel neuron-coverage metric to automatically detect class-level violations (confusion and bias errors) in DNN-based visual recognition models for image classification.
- We implemented our metric and underlying techniques in DeepInspect.
- We evaluated DeepInspect and found many errors in widely-used DNN models with precision up to 100% (avg. 72.6%) for confusion errors and up to 84.3% (avg. 66.8%) for bias errors.

Our code is available at `https://github.com/ARiSE-Lab/DeepInspect`. The errors reported by DeepInspect are available at: `https://www.ariselab.info/deepinspect`.

5.2 Methodology

We give a detailed technical description of DeepInspect. We describe a typical scenario where we envision our tool might be used in the following and design the methodology accordingly.

Usage Scenario. Similar to customer testing of post-release software, DeepInspect works in a real-world setting where a customer gets a pre-trained model and tests its performance in a sample production scenario before deployment. The customer has white-box access to the model to profile, although all the data in the production system can be *unlabeled*. In the absence of ground truth labels, the classes are defined by the *predicted labels*. These predicted labels are used as class references as DeepInspect tries to detect confusion and bias errors among the classes. DeepInspect tracks the activated neurons per class and reports a potential class-level violation if the class-level activation-patterns are too similar between two classes. Such reported errors will help customers evaluate how much they can trust the model's results related to the affected classes. As elaborated in Section 5.6, once these errors are reported back to the developers, they can focus their debugging and fixing effort on these classes. Figure 5.1 shows the DeepInspect workflow.

5.2.1 Definitions

Before we describe DeepInspect's methodology in detail, we introduce definitions that we use in the rest of the paper. The following table shows our notation.

All neurons set	$N = \{N_1, N_2, ...\}$
Activation function	$out(N, c)$ returns output for neuron N , input c.
Activation threshold	Th

Neural-Path (NP). For an input image c, we define *neural-path* as a sequence of neurons that are activated by c.

Neural-Path per Class (NP_C). For a class C_i, this metric represents a set consisting of the union of neural-paths activated by all the inputs in C_i.

For example, consider a class `cow` containing two images: a brown cow and a black cow. Let's assume they activate two neural-paths: $[N_1, N_2, N_3]$ and $[N_4, N_5, N_3]$. Thus, the neural-paths for class `cow` would be $NP_{cow} = \{[N_1, N_2, N_3], [N_4, N_5, N_3]\}$. NP_{cow} is further represented by a

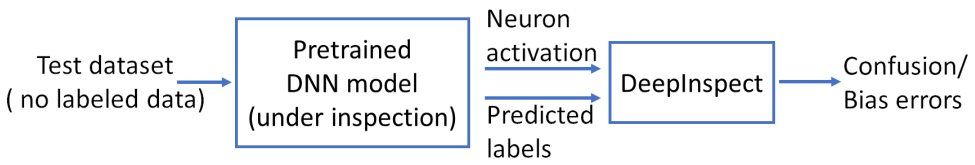


Figure 5.1: **DeepInspect Workflow**

vector $(N_1^1, N_2^1, N_3^2, N_4^1, N_5^1)$, where the superscripts represent the number of times each neuron is activated by C_{cow}. Thus, each class C_i in a dataset can be expressed with a *neuron activation frequency vector*, which captures how the model interacts with C_i.

Neuron Activation Probability: Leveraging how *frequently* a neuron N_j is activated by all the members from a class C_i, this metric estimates the probability of a neuron N_j to be activated by C_i. Thus, we define: $P(N_j \mid C_i) = \dfrac{|\{c_{ik} \mid \forall c_{ik} \in C_i, out(N_j, c_{ik}) > Th\}|}{|C_i|}$

We then construct a $n \times m$ dimensional *neuron activation probability matrix*, ρ, (n is the number of neurons and m is the number of classes) with its ij-th entry being $P(N_j \mid C_i)$.

$$\rho = \begin{array}{c} \\ N_1 \\ \dots \\ N_j \\ \dots \\ N_n \end{array} \begin{array}{c} \begin{array}{cccccc} C_1 & \dots & C_i & \dots & C_m \end{array} \\ \begin{pmatrix} p_{11} & & & & p_{1m} \\ \dots & & & & \\ p_{j1} & & \dots & & p_{jm} \\ \dots & & & & \\ p_{n1} & & & & p_{nm} \end{pmatrix} \end{array} \tag{5.1}$$

This matrix captures how a model interacts with a set of input data. The column vectors ($\rho_{\alpha m}$) represent the interaction of a class C_m with the model. Note that, in our setting, Cs are predicted labels.

Since Neuron Activation Probability Matrix (ρ) is designed to represent each class, it should be able to distinguish between different Cs. Next, we use this metric to find two different classes of errors often found in DNN systems: *confusion* and *bias* (see Table 5.1).

5.2.2 Finding Confusion Errors

In an object classification task, when the model cannot distinguish one object class from another, confusion occurs. For example, as shown in Table 5.1, a Google photo app model confuses a skier with the mountain. Thus, finding confusion errors means checking how well the model can distinguish between objects of different classes. An error happens when the model under test classifies an object with a wrong class, or for multi-label classification task, predicts two classes but only one of them is present in the test image.

We argue that the model makes these errors because during the training process the model has not learned to distinguish well between the two classes, say a and b. Therefore, the neurons activated by these objects are similar and the column vectors corresponding to these classes: $\rho_{\alpha a}$ and $\rho_{\alpha b}$ will be very close to each other. Thus, we compute the confusion score between two classes as the euclidean distance between their two probability vectors:

$$\text{NAPVD}(a,b)=\Delta(a,b)=||\rho_{\alpha a}-\rho_{\alpha b}||_2 = \sqrt{\sum_{i=1}^{n} \left(P(N_i|a) - P(N_i|b)\right)^2} \tag{5.2}$$

If the Δ value is less than some pre-defined threshold (conf_th) for two pairs of classes, the model will potentially make mistakes in distinguishing one from another, which results in confusion errors. This Δ is called NAPVD (Neuron Activation Probabiliy Vector Distance).

5.2.3 Finding Bias Errors

In an object classification task, bias occurs if the model under test shows disparate outcomes between two related classes. For example, we find that ResNet-34 pretrained by imSitu dataset, often mis-classifies a man with a baby as `woman`. We observe that in the embedded matrix ρ, $\Delta(baby, woman)$ is much smaller than $\Delta(baby, man)$. Therefore, during testing, whenever the model finds an image with a baby, it is biased towards associating the baby image with a woman. Based on this observation, we propose an inter-class distance based metric to calculate the bias

learned by the model. We define the bias between two classes a and b over a third class c as follows:

$$bias(a, b, c) := \frac{|\Delta(c, a) - \Delta(c, b)|}{\Delta(c, a) + \Delta(c, b)} \tag{5.3}$$

If a model treats objects of classes a and b similarly under the presence of a third object class c, a and b should have similar distance *w.r.t.* c in the embedded space ρ; thus, the numerator of the above equation will be small. Intuitively, the model's output can be more influenced by the nearer object classes, *i.e.* if a and b are closer to c. Thus, we normalize the disparity between the two distances to increase the influence of closer classes.

This bias score is used to measure how differently the given model treats two classes in the presence of a third object class. An **average bias** (abbreviated as avg_bias) between two objects a and b for all class objects O is defined as:

$$avg\_bias(a, b) := \frac{1}{|O| - 2} \sum_{c \in O, c \neq a, b} bias(a, b, c) \tag{5.4}$$

The above score captures the overall bias of the model between two classes. If the bias score is larger than some pre-defined threshold, we report potential *bias errors*.

Note that, even when the two classes a and b are not confused by the model, *i.e.* $\Delta(a, b) > conf\_th$, they can still show bias *w.r.t.* another class, say c, if $\Delta(a, c)$ is very different from $\Delta(b, c)$. Thus, bias and confusion are two separate types of class-level errors that we intend to study in this work.

Using these above equations we develop a novel testing tool, DeepInspect, to inspect a DNN implementing image classification tasks and look for potential confusion and bias errors. We implemented DeepInspect in the Pytorch deep learning framework and Python 2.7. All our experiments were run on Ubuntu 18.04.2 with two TITAN Xp GPUs. For all of our experiments, we set the activation threshold Th to be 0.5 for all datasets and models. We discuss why we choose 0.5 as neuron activation threshold and how different thresholds affect our performance in the section 5.6.

Table 5.2: **Study Subjects**

Dataset			Model			
Classification Task	**Name**	**#classes**	**CNN Models**	**#Neurons**	**#Layers**	**Reported Accuracy**
Multi-label classification	COCO [105]	80	ResNet-50[83]	26,560	53 Conv	0.73*
	COCO gender[83]	81	ResNet-50[209]	26,560	53 Conv	0.71*
	imSitu[106]	205,095	ResNet-34[106]	8,448	36 Conv	0.37†
Single-label classification	CIFAR-100[103]	100	CNN[245]	2,916	26	0.74
	Robust CIFAR-10[103]	10	Small CNN[244]	158	8	0.69
			Large CNN[244]	1,226	14	0.73
			ResNet[244]	1,410	34	0.70
	ImageNet[104]	1000	ResNet-50[211]	26,560	53 Conv	0.75

* reported in mean average precision, †reported in mean accuracy

5.3 Experimental Design

5.3.1 Study Subjects

We apply DeepInspect for both multi-label and single-label DNN-based classifications. Under different settings, DeepInspect automatically inspects 8 DNN models for 6 datasets. Table 5.2 summarizes our study subjects. All the models we used are standard, widely-used models for each dataset. We used pre-trained models as shown in the Table for all settings except for COCO with gender. For COCO with gender model, we used the gender labels from [83] and trained the model in the same way as [83]. imSitu model is a pre-trained ResNet-34 model [106]. There are in total 11,538 entities and 1,788 roles in the imSitu dataset. When inspecting a model trained using imSitu, we only considered the top 100 frequent entities or roles in the test dataset.

Among the 8 DNN models, three are pre-trained relatively more robust models that are trained using adversarial images along with regular images. These models are pre-trained by provably robust training approach proposed by [244]. Three models with different network structures are trained using the CIFAR10 dataset [244].

5.3.2 Constructing Ground Truth (GT) Errors

To collect the ground truth for evaluating DeepInspect, we refer to the test images misclassified by a given model. We then aggregate these misclassified image instances by their real and predicted

class-labels and estimate pair-wise confusion/bias.

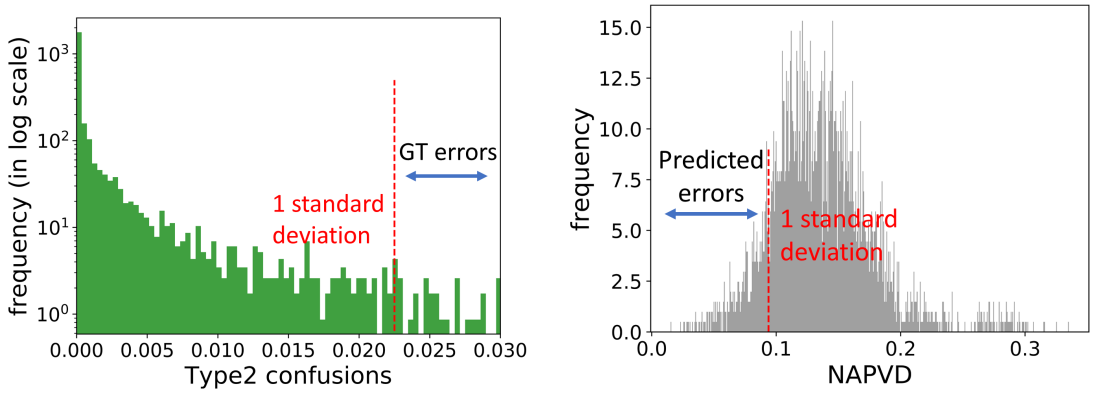
GT of Confusion Errors

Confusion occurs when a DNN often makes mistakes in disambiguating members of two different classes. In particular, if a DNN is confused between two classes, the classification error rate is higher between those two classes than between the rest of the class-pairs. Based on this, we define two types of confusion errors for single-label classification and multi-label classification separately:

Type1 confusions: In single-label classification, Type1 confusion occurs when an object of class x (*e.g.,*`violin`) is misclassified to another class y (*e.g.,*`cello`). For all the objects of class x and y, it can be quantified as: $\text{type1conf}(x, y) = \text{mean}(\text{P}(x|y), \text{P}(y|x))$ —DNN's probability to misclassify class y as x and vice-versa, and takes the average value between the two. For example, given two classes `cello` and `violin`, type1conf estimates the mean probability of `violin` misclassified to `cello` and vice versa. Note that, this is a bi-directional score, *i.e.* misclassification of y as x is the same as misclassification of x as y.

Type2 confusions: In multi-label classification, Type2 confusion occurs when an input image contains an object of class x (*e.g.,*`mouse`) and no object of class y (*e.g.,*`keyboard`), but the model predicts both classes (see Figure 5.7. For a pair of classes, this can be quantified as: $\text{type2conf}(x, y) = \text{mean}(\text{P}((x, y)|x), \text{P}((x, y)|y))$ to compute the probability to detect two objects in the presence of only one. For example, given two classes `keyboard` and `mouse`, type2conf estimates the mean probability of `mouse` being predicted while predicting `keyboard` and vice versa. This is also a bi-directional score.

We measure type1conf and type2conf by using a DNN's *true classification error* measured on a set of test images. They create the DNN's true confusion characteristics between all possible class-pairs. We then draw the distributions of type1conf and type2conf. For example, Figure 5.2a shows type2conf distribution for COCO . The class-pairs with confusion scores greater than 1 standard deviation from the mean-value are marked as pairs truly confused by the model and form



(a) Confusions distribution

(b) NAPVD distribution

Figure 5.2: **Identifying Type2 confusions for multi-classification applications. LHS shows how we marked the ground truth errors based on Type2 confusion score. RHS shows DeepInspect's predicted errors based on NAPVD score.**

our ground truth for confusion errors. For example, in the COCO dataset, there are 80 classes and thus 3160 class pairs (80*79/2); 178 class-pairs are ground-truth confusion errors.

Note that, unlike how a bug/error is defined in traditional software engineering, our suspicious confusion pairs have an inherent probabilistic nature. For example, even if a and b represent a confusion pair, it does not mean that all the images containing a or b will be misclassified by the model. Rather, it means that compared with other pairs, images containing a or b tend to have a higher chance to be misclassified by the model.

GT of Bias Errors

A DNN model is *biased* if it treats two classes differently. For example, consider three classes: `man`, `woman`, and `surfboard`. An unbiased model should not have different error rates while classifying `man` or `woman` in the presence of `surfboard`. To measure such bias formally, we define **confusion disparity** (cd) to measure differences in error rate between classes x and z and between y and z: $\text{cd}(x, y, z) = |error(x, z) - error(y, z)|$, where the error measure can be either type1conf or type2conf as defined earlier. cd essentially estimates the disparity of the model's error between classes x, y (*e.g.,* `man`, `woman`) *w.r.t.* a third class z (*e.g.,* `surfboard`).

We also define an aggregated measure **average confusion disparity** (**avg_cd**) between two

classes x and y by summing up the bias between them over all third classes and taking the average:

$$\text{avg\_cd}(x, y) := \frac{1}{|O| - 2} \sum_{z \in O, z \neq x, y} \text{cd}(x, y, z).$$

Depending on the error types we used to estimate avg_cd, we refer to $Type1$_avg_cd and $Type2$_avg_cd. We measure avg_cd using the true classification error rate reported for the test images. Similar to confusion errors, we draw the distribution of avg_cd for all possible class pairs and then consider the pairs as *truly biased* if their avg_cd score is higher than one standard deviation from the mean value. Such truly biased pairs form our ground truth for bias errors.

5.3.3 Evaluating DeepInspect

We evaluate DeepInspect using a set of test images.

Error Reporting. DeepInspect reports confusion errors based on NAPVD (see Equation (5.2)) scores—lower NAPVD indicates errors. We draw the distributions of NAPVDs for all possible class pairs, as shown in Figure 5.2b. Class pairs having NAPVD scores lower than 1 standard deviation from the mean score are marked as potential confusion errors.

As discussed in Section 5.2.3, DeepInspect reports bias errors based on avg_bias score (see Equation (5.4)), where higher avg_bias means class pairs are more prone to bias errors. Similar to above, from the distribution of avg_bias scores, DeepInspect predicts pairs with avg_bias greater than 1 standard deviation from the mean score to be erroneous. Note that, while calculating error disparity between classes a, b *w.r.t.* c (see Equation (5.3)), if both a and b are far from c in the embedded space ρ, disparity of their distances (Δ) should not reflect true bias. Thus, while calculating avg_bias(a, b) we further filter out the triplets where $\Delta(c, a) > th \wedge \Delta(c, b) > th$, where th is some pre-defined threshold. In our experiment, we remove all the class-pairs having Δ larger than 1 standard deviation (*i.e.* th) from the mean value of all $Delta$s across all the class-pairs.

Evaluation Metric. We evaluate DeepInspect in two ways:

Precision & Recall. We use precision and recall to measure DeepInspect's accuracy. For each

error type t, suppose that E is the number of errors detected by DeepInspect and A is the the number of true errors in the ground truth set. Then the precision and recall of DeepInspect are $\frac{|A \cap E|}{|E|}$ and $\frac{|A \cap E|}{|A|}$ respectively.

Area Under Cost Effective Curve (AUCEC). Similarly to how static analysis warnings are ranked based on their priority levels [246], we also rank the erroneous class-pairs identified by DeepInspect based on the decreasing order of error proneness, *i.e.* most error-prone pairs will be at the top. To evaluate the ranking we use a cost-effectiveness measure [247], AUCEC (Area Under the Cost-Effectiveness Curve), which has become standard to evaluate rank-based bug-prediction systems [248, 249, 250, 246, 251].

Cost-effectiveness evaluates when we inspect/test top n% class-pairs in the ranked list (*i.e.* inspection cost), how many true errors are found (*i.e.* effectiveness). Both cost and effectiveness are normalized to 100%. Figure 5.6 shows cost on the x-axis, and effectiveness on the y-axis, indicating the portion of the ground truth errors found. AUCEC is the area under this curve.

Baseline. We compare DeepInspect *w.r.t.* two baselines:

(i) MODE-inspired: A popular way to inspect each image is to inspect a feature vector, which is an output of an intermediate layer [65, 46]. However, abstracting a feature vector per image to the class level is non-trivial. Instead, for a given layer, one could inspect the weight vector ($w_l = [w_l^0, w_l^1, ..., w_l^n]$) of a class, say l, where the superscripts represent a feature. Similar weight-vectors are used in MODE [65] to compare the difference in feature importance between two image groups. In particular, from the last linear layer before the output layer we extract such per-class weight vectors and compute the pairwise distances between the weight vectors. Using these pairwise distances we calculate confusion and bias metrics as described in Section 5.2.

(ii) Random: We also build a random model that picks random class-pairs for inspection [252] as a baseline.

For AUCEC evaluation, we further show the performance of an optimal model that ranks the class-pairs perfectly—if $n\%$ of all the class-pairs are truly erroneous, the optimal model would rank them at the top such that with lower inspection budget most of the errors will be detected.

The optimal curve gives the lower upper bound of the ranking scheme.

Research Questions. With this experimental setting, we investigate the following three research questions to evaluate DeepInspect for DNN image classifiers:

- **RQ1.** Can DeepInspect distinguish between different classes?
- **RQ2.** Can DeepInspect identify the confusion errors?
- **RQ3.** Can DeepInspect identify the bias errors?

5.4 Results

We begin our investigation by checking whether de-facto neuron coverage-based metrics can capture class separation.

RQ1. Can DeepInspect distinguish between different classes?

Motivation. The heart of DeepInspect's error detection technique lies in the fact that the underlying Neuron Activation Probability metric (ρ) captures each class abstraction reasonably well and thus distinguishes between classes that do not suffer from class-level violations. In this RQ we check whether this is indeed true. We also check whether a new metric ρ is necessary, *i.e.*, whether existing neuron-coverage metrics could capture such class separations.

Approach. We evaluate this RQ *w.r.t.* the training data since the DNN behaviors are not tainted with inaccuracies associated with the test images. Thus, all the class-pairs are benign. We evaluate this RQ in three settings: (i) using DeepInspect's metrics, (ii) neuron-coverage proposed by Pei *et al.* [44], and (iii) other neuron-activation related metrics proposed by DeepGauge [50].

Setting-1. DeepInspect. Our metric, Neuron Activation Probability Matrix (ρ), by construction is designed per class. Hence it would be unfair to directly measure its capability to distinguish between different classes. Thus, we pose this question in slightly a different way, as described below. For multi-label classification, each image contains multiple class-labels. For example, an image might have labels for both `mouse` and `keyboard`. Such coincidence of labels may create

confusion—if two labels always appear together in the ground truth set, no classifier can distinguish between them. To check how many times two labels coincide, we define a coincidence score between two labels L_a and L_b as: $coincidence\ (L_a, L_b) = mean(P\ (L_a, L_b|L_a)\ , P\ (L_a, L_b|L_b))$.

The above formula computes the minimum probability of labels L_a and L_b occurring together in an image given that one of them is present. Note that this is a bi-directional score, *i.e.* we treat the two labels similarly. The *mean* operation ensures we detect the least coincidence in either direction. A low value of coincidence score indicates two class-labels are easy to separate and vice versa.

Now, to check DeepInspect's capability to capture class separation, we simply check the correlation between coincidence score and confusion score (NAPVD) from Equation 5.2 for all possible class-label pairs. Since only multi-label objects can have label coincidences, we perform this experiment for a pre-trained ResNet-50 model on the COCO multi-label classification task.

A Spearman correlation coefficient between the confusion and coincidence scores reaches a value as high as 0.96, showing strong statistical significance. The result indicates that DeepInspect can disambiguate most of the classes that have a low confusion scores.

Interestingly, we found some pairs where coincidence score is high, but DeepInspect was able to isolate them. For example, (`cup`, `chair`), (`toilet`, `sink`), *etc.*. Manually investigating such cases reveals that although these pairs often appear together in the input images, there are also enough instances when they appear by themselves. Thus, DeepInspect disambiguates between these classes and puts them apart in the embedded space ρ. These results indicate DeepInspect can also learn some hidden patterns from the context and, thus, can go beyond inspecting the training data coincidence for evaluating model bias/confusion, which is the de facto technique among machine learning researchers [83].

Next, we investigate whether popular white-box metrics can distinguish between different classes.

Setting-2. Neuron Coverage (*NC*) [44] computes the ratio of the union of neurons activated by an input set and the total number of neurons in a DNN. Here we compute *NC* per class-label, *i.e.*

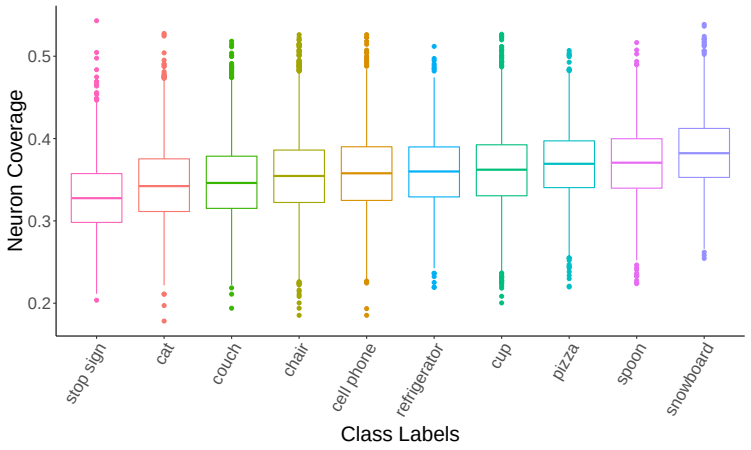


Figure 5.3: **Distribution of neuron coverage per class label, for 10 randomly picked class labels, from the COCO dataset.**

for a given class-label, we measure the number of neurons activated by the images tagged with that label *w.r.t.* to the total neurons. The activation threshold we use is 0.5. We perform this experiment on COCO and CIFAR-100 to study multi- and single-label classifications. Figure 5.3 shows results for COCO . We observe similar results for CIFAR-100 .

Each boxplot in the figure shows the distribution of neuron coverage per class-label across all the relevant images. These boxplots visually show that *different labels* have very *similar NC* distribution. We further compare these distributions using Kruskal Test [253], which is a non-parametric way of comparing more than two groups. Note that we choose a non-parametric measure as *NC*s may not follow normal distributions. (Kruskal Test is a parametric equivalent of the one-way analysis of variance (ANOVA).) The result reports a $p - value << 0.05$, *i.e.* some differences exist across these distributions. However, a pairwise Cohend's effect size for each class-label pair, as shown in the following table, shows more than 56% and 78% class-pairs for CIFAR-100 and COCO have small to negligible effect size. This means neuron coverage cannot reliably distinguish a majority of the class-labels.

Effect Size of neuron coverage across different classes				
Exp Setting	negligible	small	medium	large
COCO	40.51%	38.19%	16.96%	4.34%
CIFAR-100	31.94%	25.69%	23.87%	18.48%

Setting-3. DeepGauge [50]. Ma *et al.* [50] argue that each neuron has a primary region of operation; they identify this region by using a boundary condition $[low, high]$ on its output during training time; outputs outside this region $((-\infty, low) \cup (high, +\infty))$ are marked as corner cases. They therefore introduce multi-granular neuron and layer-level coverage criteria. For neuron coverage they propose: (i) *k-multisection coverage* to evaluate how thoroughly the primary region of a neuron is covered, (ii) *boundary coverage* to compute how many corner cases are covered, and (iii) *strong neuron activation coverage* to measure how many corner case regions are covered in $(high, +\infty)$ region. For layer-level coverage, they define (iv) *top-k neuron coverage* to identify the most active k-neurons for each layer, and (v) *top-k neuron pattern* for each test-case to find a sequence of neurons from the top-k most active neurons across each layer.

We investigate whether each of these metrics can distinguish between different classes by measuring the above metrics for individual input classes following Ma *et al.*'s methodology. We first profiled every neuron upper- and lower-bound for each class using the training images containing that class-label. Next, we computed per-class neuron coverage using test images containing that class; for k-multisection coverage we chose $k = 100$ to scale up the analysis. It should be noted that we also tried $k = 1000$ (which is used in the original DeepGauge paper) and observed similar results (not shown here).

For layer-level coverage, we directly used the input images containing each class, where we select $k = 1$.

Figure 5.4 shows the results as a histogram of the above five coverage criteria for the COCO dataset. For all five coverage criteria, there are many class-labels that share similar coverage. For example, in COCO , there are 52 labels with k-multisection neuron coverage with values between 0.31 and 0.32. Similarly, there are 40 labels with 0 neuron boundary coverage. Therefore, none of the five coverage criteria are an effective way to distinguish between different equivalence classes. The same conclusion was drawn for the CIFAR-100 dataset.

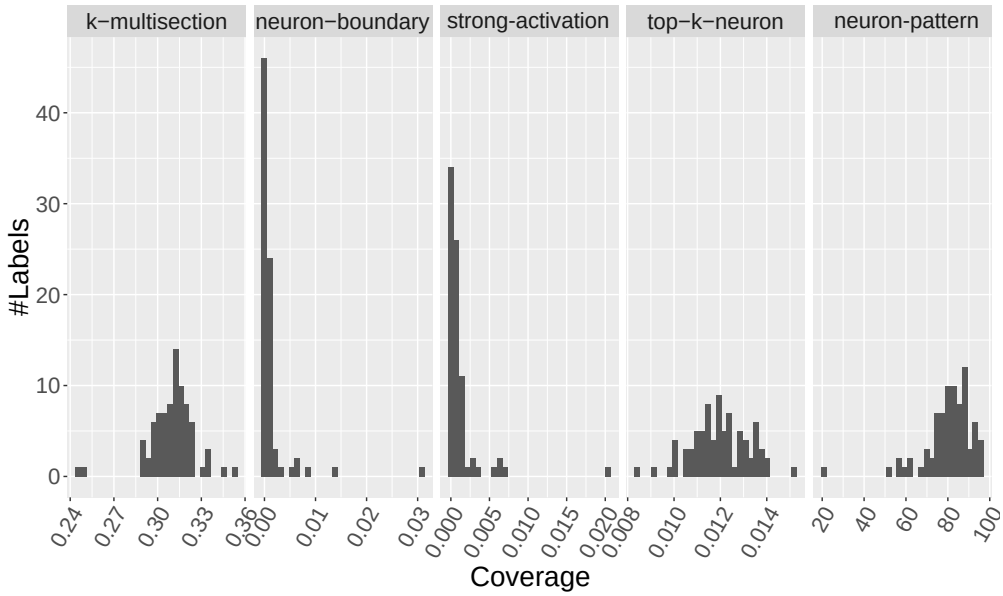


Figure 5.4: **Histogram of DeepGauge [50] multi-granular coverage per class label for COCO dataset**

> **Result 1:** *DeepInspect can disambiguate classes better than previous coverage-based metrics for the image classification task.*

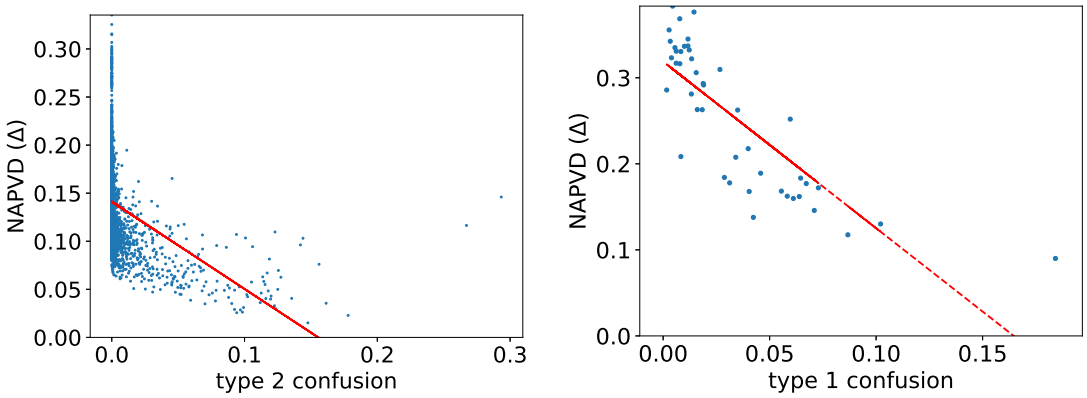
We now investigate DeepInspect's capability in detecting confusion and bias errors in DNN models.

RQ2. Can DeepInspect identify the confusion errors?

Motivation. To evaluate how well DeepInspect can detect class-level violations, in this RQ, we report DeepInspect's ability to detect the first type of violation, *i.e.*, Type1/Type2 confusions *w.r.t.* to ground truth confusion errors, as described in Section 5.3.2.

We first explore the correlation between NAPVD and ground truth Type1/Type2 confusion score. Strong correlation has been found for all 8 experimental settings. Figure 5.5 gives examples on COCO and CIFAR-10. These results indicate that NAPVD can be used to detect confusion errors—lower NAPVD means more confusion.

Approach. By default, DeepInspect reports all the class-pairs with NAPVD scores one standard deviation less than the mean NAPVD score as error-prone (See Figure 5.2b). In this setting, as the result shown on Table 5.3, DeepInspect reports errors at high recall under most settings. Specif-



(a) COCO dataset + ResNet-50 (b) Robust CIFAR-10 Small

Figure 5.5: **Strong negative Spearman correlation (-0.55 and -0.86) between NAPVD and ground truth confusion scores.**

ically, on CIFAR-100 and robust CIFAR-10 ResNet, DeepInspect can report errors as high as 71.8%, and 100%, respectively. DeepInspect has identified thousands of confusion errors.

If higher precision is wanted, a user can choose to inspect only a small set of confused pairs based on NAPVD. As also shown in Table 5.3, when only the top1% confusion errors are reported, a much higher precision is achieved for all the datasets. In particular, DeepInspect identifies 31 and 39 confusion errors for the COCO model and the CIFAR-100 model with 100% and 79.6% precision, respectively. The trade-off between precision and recall can be found on the cost-effective curves shown on Figure 5.6, which show overall performance of DeepInspect at different inspection cutoffs. Overall, *w.r.t.* a random baseline mode, DeepInspect is gaining AUCEC performance from 61.6% to 85.7%; *w.r.t.* a MODE baseline mode, DeepInspect is gaining AUCEC performance from 10.2% to 28.2%.

Figure 5.7 and Figure 5.8 give some specific confusion errors found by DeepInspect in the COCO and the ImageNet settings. In particular, as shown in Figure 5.7a, when there is only a keyboard but no mouse in the image, the COCO model reports both. Similarly, Figure 5.8a shows confusion errors on (cello, violin). There are several cellos in this image, but the model predicts it to show a violin.

Across all three relatively more robust CIFAR-10 models DeepInspect identifies (cat, dog),

Table 5.3: **DeepInspect performance on detecting confusion errors**

		NAPVD < mean-1std				Top 1%			
		TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
COCO	DeepInspect	138	256	0.350	0.775	31	0	1	0.174
	MODE	126	382	0.248	0.708	26	5	0.839	0.146
	random	22	372	0.056	0.124	1	30	0.032	0.006
COCO gender	DeepInspect	139	286	0.327	0.827	32	0	1	0.190
	MODE	125	379	0.248	0.744	30	2	0.938	0.179
	random	22	403	0.052	0.131	1	31	0.031	0.006
CIFAR-100	DeepInspect	206	584	0.261	0.718	39	10	0.796	0.136
	MODE	111	605	0.155	0.387	22	27	0.449	0.077
	random	45	745	0.057	0.157	2	47	0.041	0.007
R CIFAR-10 S	DeepInspect	4	6	0.400	0.800	-	-	-	-
	MODE	3	4	0.429	0.600	-	-	-	-
	random	1	9	0.100	0.200	-	-	-	-
R CIFAR-10 L	DeepInspect	3	4	0.430	0.600	-	-	-	-
	MODE	3	5	0.375	0.600	-	-	-	-
	random	0	7	0	0	-	-	-	-
R CIFAR-10 R	DeepInspect	5	3	0.625	1	-	-	-	-
	MODE	1	3	0.250	0.200	-	-	-	-
	random	0	8	0	0	-	-	-	-
ImageNet	DeepInspect	4014	69957	0.054	0.617	1073	3922	0.215	0.165
	MODE	3428	66987	0.049	0.527	1591	3404	0.319	0.245
	random	962	73009	0.013	0.148	65	4930	0.013	0.010
imSitu	DeepInspect	48	58	0.453	0.165	31	19	0.620	0.107
	random	6	100	0.057	0.020	2	48	0.040	0.007

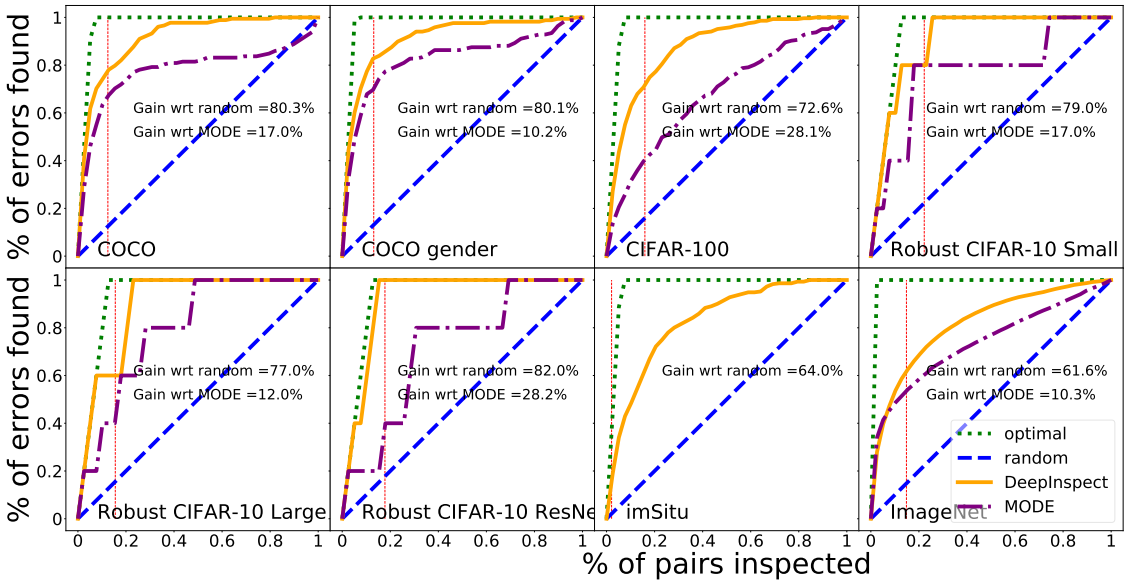


Figure 5.6: **AUCEC plot of Type1/Type2 Confusion errors in three different settings. The red vertical line marks 1-standard deviation less from mean NAPVD score. DeepInspect marks all class-pairs with NAPVD scores less than the red mark as potential errors.**

(a) (keyboard,mouse)

(b) (oven,microwave)

Figure 5.7: **Confusion errors identified in COCO model. In each pair the second object is mistakenly identified by the model.**

(a) (cello, violin)

(b) (library, bookshop)

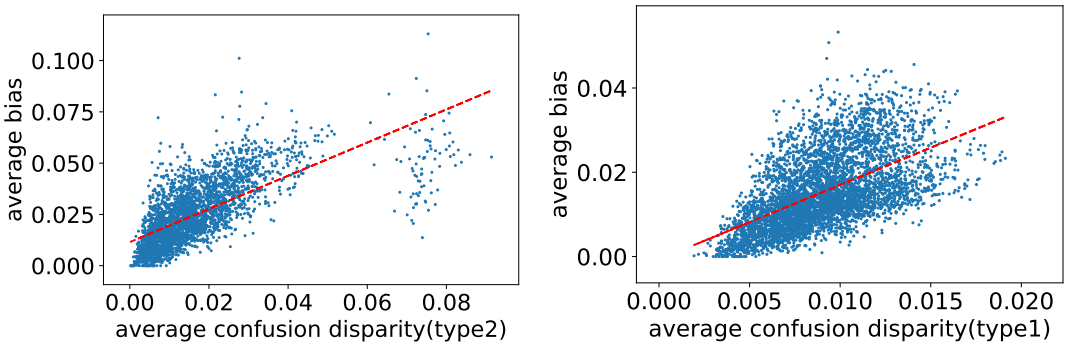
Figure 5.8: **Confusion errors identified in the ImageNet model. For each pair, the second object is mistakenly identified by the model.**

(bird, deer) and (automobile, truck) as buggy pairs, where one class is very likely to be mistakenly classified as the other class of the pair. This indicates that these confusion errors are to be tied to the training data, so all the models trained on this dataset including the robust models may have these errors. These results further show that the confusion errors are orthogonal to the norm-based adversarial perturbations and we need a different technique to address them.

We also note that the performance of all methods degrades quite a bit on ImageNet. ImageNet is known to have a complex structure, and all the tasks, including image classification and robust image classification [254] usually have inferior performance compared with simpler datasets like CIFAR-10 or CIFAR-100. Due to such inherent complexity, the class representation in the embedded space is less accurate, and thus the relative distance between two classes may not correctly reflect a model's confusion level between two classes.

Result 2: *DeepInspect can successfully find confusion errors with precision 21% to 100% at top1% for both single- and multi-object classification tasks. DeepInspect also finds confusion errors in robust models.*

RQ3. Can DeepInspect identify the bias errors?



(a) COCO (b) CIFAR-100

Figure 5.9: **Strong positive Spearman's correlation (0.76 and 0.62) exist between avg_cd and avg_bias while detecting classification bias.**

Motivation. To assess DeepInspect's ability to detect class-level violations, in this RQ, we report DeepInspect's performance in detecting the second type of violation, *i.e.*, Bias errors as described

in Section 5.3.2.

Approach. We evaluate this RQ by estimating a model's bias (avg_bias) using Equation (5.4) *w.r.t.* the ground truth (avg_cd), computed as in Section 5.3.2. We first explore the correlation between pairwise avg_cd and our proposed pairwise avg_bias; Figure 5.9 shows the results for COCO and CIFAR-10. Similar trends were found in the other datasets we studied. The results show that a strong correlation exists between avg_cd and avg_bias. In other words, our proposed avg_bias is a good proxy for detecting confusion errors.

Table 5.4: **DeepInspect performance on detecting bias errors**

		avg_bias > mean+1std				Top 1%			
		TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
COCO	DeepInspect	249	278	0.472	0.759	24	8	0.75	0.073
	MODE	145	324	0.309	0.442	12	20	0.375	0.037
	random	54	472	0.103	0.167	3	28	0.103	0.010
COCO gender	DeepInspect	218	325	0.401	0.568	17	16	0.515	0.044
	MODE	151	328	0.315	0.393	13	20	0.394	0.034
	random	64	478	0.118	0.168	3	28	0.118	0.010
CIFAR-100	DeepInspect	310	543	0.363	0.380	29	21	0.580	0.036
	MODE	69	315	0.180	0.085	5	45	0.100	0.001
	random	140	711	0.165	0.172	8	41	0.165	0.010
R CIFAR-10 S	DeepInspect	7	4	0.636	0.778	-	-	-	-
	MODE	3	10	0.231	0.333	-	-	-	-
	random	2	8	0.200	0.222	-	-	-	-
R CIFAR-10 L	DeepInspect	6	7	0.462	0.667	-	-	-	-
	MODE	8	14	0.364	0.889	-	-	-	-
	random	2	9	0.200	0.267	-	-	-	-
R CIFAR-10 R	DeepInspect	6	3	0.667	0.667	-	-	-	-
	MODE	8	14	0.364	0.889	-	-	-	-
	random	1	7	0.200	0.200	-	-	-	-
ImageNet	DeepInspect	26704	48913	0.353	0.330	3253	1742	0.651	0.040
	MODE	23881	47503	0.335	0.295	2355	2640	0.471	0.029
	random	12234	63381	0.162	0.151	808	4186	0.162	0.010
imSitu	DeepInspect	408	311	0.567	0.718	43	8	0.843	0.076
	random	80	638	0.112	0.142	5	44	0.112	0.010

As in RQ2, we also do a precision-recall analysis *w.r.t.* finding the bias errors across all the datasets. We analyze the precision and recall of DeepInspect when reporting bias errors at the cutoff Top1%(avg_bias) and mean(avg_bias)+standard deviation(avg_bias), respectively. The results are shown in Table 5.4. At cutoff Top1%(avg_bias), DeepInspect detects suspicious pairs with precision as high as 75% and 84% for COCO and imSitu, respectively. At cutoff mean(avg_bias)+standard deviation(avg_bias), DeepInspect has high recall but lower precision:

DeepInspect detects ground truth suspicious pairs with recall at 75.9% and 71.8% for COCO and imSitu. DeepInspect can report 657(=249+408) total true bias bugs across the two models. DeepInspect outperforms the random baseline by a large margin at both cutoffs. As in the case of detecting confusion errors, there is a significant trade-off between precision and recall. This can be customized based on user needs. The cost-effectiveness analysis in Figure 5.10 shows the entire spectrum.

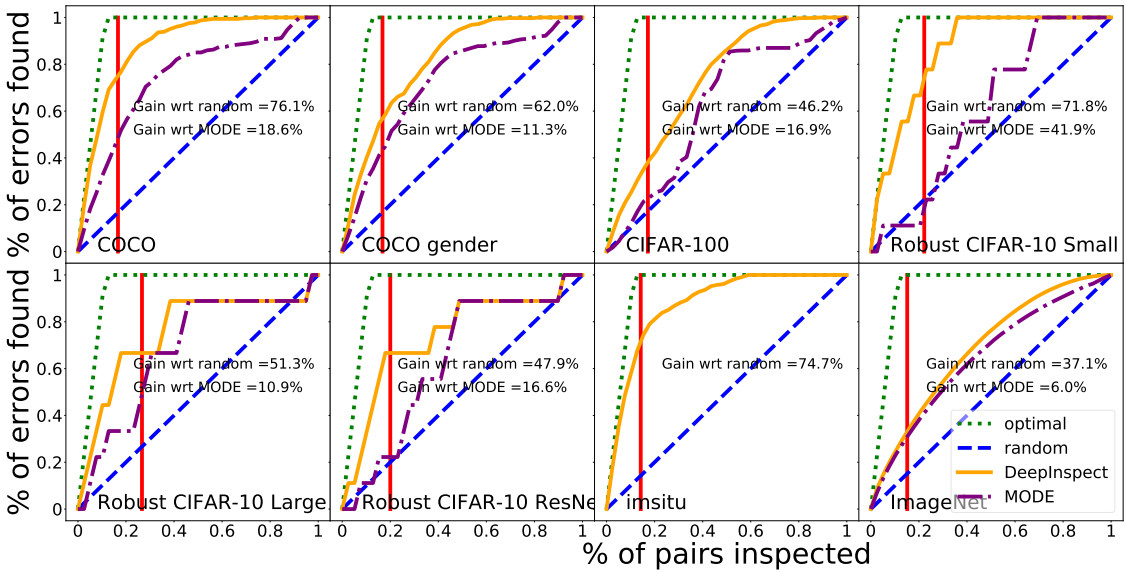


Figure 5.10: **Bias errors detected *w.r.t.* the ground truth of avg_cd beyond one standard deviation from mean.**

As shown in Figure 5.10, DeepInspect outperforms the baseline by a large margin. The AUCEC gains of DeepInspect are from 37.1% to 76.1% w.r.t. the random baseline and from 6.0% to 41.9% w.r.t. the MODE baseline across the 8 settings. DeepInspect's performance is close to the optimal curve under some settings, specifically the AUCEC gains of the optimal over DeepInspect are only 7.11% and 7.95% under the COCO and ImSitu settings, respectively.

Inspired by [83], which shows bias exists between men and women in COCO for the gender image captioning task, we analyze the most biased third class c for a and b being men and women. As shown in Figure 5.11, we found that sports like skiing, snowboarding, and surfboarding are more closely associated with men and thus misleads the model to predict the women in the images as men. Figure 5.12 shows results on imSitu, where we found that the model tends to associate the

class “inside” with women while associating the class “outside” with men.

Figure 5.11: **The model classifies the women in these pictures as men in the COCO dataset.**

We generalize the idea by choosing classes *a* and *b* to be any class-pair. We found that similar bias also exists in the single-label classification settings. For example, in ImageNet, one of the highest biases is between Eskimo_dog and rapeseed *w.r.t.* Siberian_husky. The model tends to confuse the two dogs but not Eskimo_dog and rapeseed. This makes sense since Eskimo_dog and Siberian_husk are both dogs so more easily misclassified by the model.

Figure 5.12: **The model classifies the man in the first figure to be a woman and the woman in the second figure to be a man.**

One of the fairness violations of a DNN system can be drastic differences in accuracy across groups divided according to some sensitive feature(s). In black-box testing, the tester can get a number indicating the degree of fairness has been violated by feeding into the model a validation set. In contrast, DeepInspect provides a new angle to the fairness violations. The neuron distance difference between two classes *a* and *b* *w.r.t.* a third class *c* sheds light on why the model tends to be more likely to confuse between one of them and *c* than the other. We leave a more comprehensive examination on interpreting bias/fairness violations for future work.

Result 3: *DeepInspect can successfully find bias errors for both single- and multi-label classification tasks, and even for the robust models, from 52% to 84% precision at top1%.*

5.5 Related Work

Software Testing & Verification of DNNs. Prior research proposed different white-box testing criteria based on neuron coverage [44, 50, 45] and neuron-pair coverage [108]. Sun *et al.* [57] presented a concolic testing approach for DNNs called DeepConcolic. They showed that their concolic testing approach can effectively increase coverage and find adversarial examples. Odena and Goodfellow proposed TensorFuzz[243], which is a general tool that combines coverage-guided fuzzing with property-based testing to generate cases that violate a user-specified objective. It has applications like finding numerical errors in trained neural networks, exposing disagreements between neural networks and their quantized versions, surfacing broken loss functions in popular GitHub repositories, and making performance improvements to TensorFlow. There are also efforts to verify DNNs [239, 67, 156, 70] against adversarial attacks. However, most of the verification efforts are limited to small DNNs and pixel-level properties. It is not obvious how to directly apply these techniques to detect class-level violations.

Adversarial Deep Learning. DNNs are known to be vulnerable to well-crafted inputs called adversarial examples, where the discrepancies are imperceptible to a human but can easily make DNNs fail [109, 110, 111, 112, 77, 113, 114, 115, 116, 117, 118, 119, 120, 121]. Much work has been done to defend against adversarial attacks [122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 75]. Our methods have potential to identify adversarial inputs. Moreover, adversarial examples are usually out of distribution data and not realistic, while we can find both out-distribution and in-distribution corner cases. Further, we can identify a general weakness or bug rather than focusing on crafted attacks that often require a strong attacker model (*e.g.,* the attacker adds noise to a stop sign image).

Interpreting DNNs. There has been much research on model interpretability and visualiza-

tion [255, 256, 257, 258, 259, 260]. A comprehensive study is presented by Lipton [255]. Dong *et al.* [260] observed that instead of learning the semantic features of whole objects, neurons tend to react to different parts of the objects in a recurrent manner. Our probabilistic way of looking at neuron activation per class aims to capture holistic behavior of an entire class instead of an individual object so diverse features of class members can be captured. Closest to ours is by Papernot *et al.* [261], who used nearest training points to explain adversarial attacks. In comparison, we analyze the DNN's dependencies on the entire training/testing data and represent it in Neuron Activation Probability Matrix. We can explain the DNN's bias and weaknesses by inspecting this matrix.

Evaluating Models' Bias/Fairness. Evaluating the bias and fairness of a system is important both from a theoretical and a practical perspective [134, 135, 136, 137]. Related studies first define a fairness criteria and then try to optimize the original objective while satisfying the fairness criteria [138, 139, 140, 141, 142, 143]. These properties are defined either at individual [138, 144, 145] or group levels [146, 139, 147]. In this work, we propose a definition of a bias error for image classification closely related to fairness notions at group-level. Class membership can be regarded as the sensitive feature and the equality that we want to achieve is for the confusion levels of two groups w.r.t. any third group. We showed the potential of DeepInspect to detect such violations.

Galhotra *et al.* [148] first applied the notion of software testing to evaluating software fairness. They mutate the sensitive features of the inputs and check whether the output changes. One major problem with their proposed method, Themis, is that it assumes the model takes into account sensitive attribute(s) during training and inference. This assumption is not realistic since most existing fairness-aware models drop input-sensitive feature(s). Besides, Themis will not work on image classification, where the sensitive attribute (*e.g.,*, gender, race) is a visual concept that cannot be flipped easily. In our work, we use a white-box approach to measure the bias learned by the model during training. Our testing method does not require the model to take into account any sensitive feature(s). We propose a new fairness notion for the setting of multi-object classification, *average confusion disparity*, and a proxy, *average bias*, to measure for any deep learning model

even when only unlabeled testing data is provided. In addition, our method tries to provide an explanation behind the discrimination. A complementary approach by Papernot *et al.* [261] shows such explainability behind model bias in a single classification setting.

5.6 Discussion & Threats to Validity

Discussion. In the literature, bug detection, debugging, and repair are usually three distinct tasks, and there is a large body of work investigating each separately. In this work, we focus on bug detection for image classifier software. A natural follow-up of our work will be debugging and repair leveraging DeepInspect's bug detection. We present some preliminary results and thoughts.

A commonly used approach to improving (*i.e.* fixing) image classifiers is active learning, which consists of adding more labeled data by smartly choosing what to label next. In our case, we can use NAPVD to identify the most confusing class pairs, and then target those pairs by collecting additional examples that contain individual objects from the confusing pairs. We download 105 sample images from Google Images that contain isolated examples of these categories so that the model learns to disambiguate them. We retrain the model from scratch using the original training data and these additional examples. Using this approach, we have some preliminary results on the COCO dataset. After retraining, we find that the type2conf of the top confused pairs reduces. For example, the type2conf(baseball bat, baseball glove) is reduced from 0.23 to 0.16, and type2conf(refrigerator, oven) is reduced from 0.14 to 0.10. Unlike traditional active learning approaches that encourage labeling additional examples near the current decision boundary of the classifier, our approach encourages the labeling of problematic examples based on confusion bugs.

Another potential direction to explore is to use DeepInspect in tandem with debugging & repair tools for DNN models like MODE [65]. DeepInspect enables the user to focus debugging effort on the vulnerable classes even in the absence of labeled data. For instance, once DeepInspect identifies the vulnerable class-pairs, one can use the GAN-based approach proposed in MODE to generate more training data from these class-pairs, apply MODE to identify the most vulnerable features in these pairs to select for retraining.

We have also explored how the neuron coverage threshold(th) used in computing $NAPVD$ affects our performance in detecting confusion and bias errors. We studied one multi-label classification task COCO and one single-label classification task CIFAR-100. Table 5.5, 5.6, 5.7, 5.8 show how our precision and recall change when using different neuron coverage thresholds (th). We observed that for CIFAR-100 and COCO that DeepInspect's accuracies are overall stable at $0.4 \leq th \leq 0.75$. With smaller th(< 0.25), too many neurons are activated pulling the per-class activation-probability-vectors closer to each other. In contrast, with higher th(> 0.75), important activation information gets lost. Thus, we select $th = 0.5$ for all the other experiments to avoid either issue.

Table 5.5: **WR impact of neuron coverage threshold on detecting confusion errors for COCO**

NC threshold	NAPVD < mean-1std				Top 1%			
	TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
0.25	36	18	0.67	0.20	23	8	0.74	0.13
0.40	150	215	0.41	0.84	31	0	1	0.17
0.50	138	256	0.35	0.78	31	0	1	0.17
0.60	137	264	0.34	0.77	30	1	0.97	0.17
0.75	135	271	0.33	0.76	29	2	0.94	0.16

Table 5.6: **WR impact of neuron coverage threshold on detecting confusion errors for CIFAR-100**

NC threshold	NAPVD < mean-1std				Top 1%			
	TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
0.25	188	629	0.23	0.66	34	15	0.69	0.12
0.40	197	550	0.26	0.69	39	10	0.80	0.14
0.50	206	584	0.26	0.72	39	10	0.80	0.14
0.60	211	596	0.26	0.74	37	12	0.76	0.13
0.75	195	604	0.24	0.68	37	12	0.76	0.13

Threats to Validity. We only test DeepInspect on 6 datasets under 8 settings. We include both single-class and multi-class as well as regular and robust models to address these threats as much as possible.

Another limitation is that DeepInspect needs to decide thresholds for both confusion errors and bias errors, and a threshold for discarding low-confusion triplets in the estimation of avg_bias.

Table 5.7: **WR impact of neuron coverage threshold on detecting bias errors for COCO**

NC threshold	avg_bias > mean+1std				Top 1%			
	TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
0.25	218	280	0.438	0.665	26	6	0.812	0.079
0.40	260	275	0.486	0.793	20	12	0.625	0.061
0.50	249	278	0.472	0.759	24	8	0.75	0.073
0.60	190	273	0.410	0.579	24	8	0.75	0.073
0.75	197	54	0.785	0.601	32	0	1	0.098
0.90	201	102	0.663	0.592	32	0	1	0.094

Table 5.8: **WR impact of neuron coverage threshold on detecting bias errors for CIFAR-100**

NC threshold	avg_bias > mean+1std				Top 1%			
	TP	**FP**	**Precision**	**Recall**	**TP**	**FP**	**Precision**	**Recall**
0.25	289	569	0.337	0.355	18	32	0.36	0.022
0.40	272	545	0.333	0.334	27	23	0.54	0.033
0.50	310	543	0.363	0.380	29	21	0.58	0.036
0.60	279	473	0.371	0.342	26	24	0.54	0.032
0.75	276	455	0.378	0.339	29	21	0.58	0.036
0.90	179	587	0.234	0.220	12	38	0.24	0.015

Instead of choosing fixed threshold, we mitigate this threat by choosing thresholds that are one standard deviation from the corresponding mean values and, also, reporting performance at top1%.

The task of accurately classifying any image is notoriously difficult. We simplify the problem by testing the DNN model only for the classes that it has seen during training. For example, while training, if a DNN does not learn to differentiate between black vs. brown `cows` (*i.e.*, all the cow images only have label cow and they are treated as belonging to the same class by the DNN), DeepInspect will not be able to test these sub-groups.

5.7 Conclusion

Our testing tool for DNN image classifiers, DeepInspect, automatically detects confusion and bias errors in classification models. We applied DeepInspect to six different popular image classification datasets and eight pretrained DNN models, including three so-called relatively more robust models. We show that DeepInspect can successfully detect class-level violations for both single-

and multi-label classification models with high precision.

Chapter 6: Repairing Group-Level Errors Using Weighted Regularization

In Chapter 5, we introduced our work DeepInspect, which is designed to identify group-level errors(confusion errors and bias errors) for DNN based software. In this chapter, we work on addressing a follow-up and more challenging problem - automatically repair DNN based software for confusion errors and bias errors. As proposed by Goodfellow *et al.*, regularization are all different approaches expressing preference for different solutions[262]. To solve this problem, we propose WR consists of five different weighted regularization techniques to make a DNN model takes more effort in learning from distinguishing target classes to reduce confusion between target pairs and bias among target triples. These five different weighted regularization techniques function at different stage of retraining or inference of DNNs including input phase, layer phase, loss phase and output phase. These different techniques make it possible for effective repairing in different scenarios. Our experimental results show that WR can effectively fix confusion and bias errors and these methods all have their pros, cons and applicable scenarios.

We publicly release the source code[1]. All images, figures, tables, equations, and text included in this chapter is based on a recently collaborative work.

6.1 Introduction

Deep Neural Networks are widely used nowadays as components in many critical applications like self-driving cars, face-recognition, medical diagnosis, *etc.* Unlike traditional software, although a DNN model has no code logics, it may still suffer from a different type of "serious bugs"[263, 78]. For example, it has been found that Google photo-tagging app tagged pictures of two dark-skinned people as “gorillas” [79]. Analogous to traditional software bugs, previous work in Software Engineering (SE) has denoted classification errors like this as *model bugs* [65] that

[1] https://github.com/deepfixdeepfix/dnnfix

arise from either biased training data, problematic model architecture, training procedure error or the combination of them.

DNN classification errors fall into two main categories, instance-wise and group-wise. The former has been well-studied in the previous literature. In essence, An instance-wise error happens when a DNN model outputs inconsistent prediction across different semantic-preserving transformations of a given input [44, 45, 76, 72]. Over the past few years, researchers have found numerous such transformations such as norm-bounded perturbation[72], natural transformation[47], or physical attack[77] to fool a well-trained DNN classifier. The fixing strategies such as adversary training, data augmentation are also widely studied [72, 47]. In contrast, group-wise error is about the DNN model's weak performance on differentiating among certain classes or has inconsistent performance across classes[78]. There are very few work on repairing group-wise errors and it only receives attentions recently [78]. This type of bugs is very concerning since it has been found to relate to many real-world notorious errors without malicious attackers[78]. Some works proposed techniques to detect this kind of errors, however until now, no fixing methods have been proposed for repairing them. To bridge this gap, in this work, we propose a generic fixing method for repairing such errors of any given DNN models.

The group-level errors definition proposed in [78] consists of two main types with different root causes: *(i)* ***Confusion****: The model cannot differentiate one class from another. For example, Google Photos confuses skier and mountain [81]. (ii)* ***Bias****: The model shows disparate outcomes between two related groups. For example, Zhao* et al. *[83] find classification bias in favor of women on activities like shopping, cooking, washing,* etc.. Figure 6.1 presents two concrete examples of both types of errors from COCO and Image-Net reported in [78]. Note that unlike an instance-level error, such group-level error affects all the images falling into the groups.

The causes of such errors can be certain classes are harder to be differentiated from each other. For example, in CIFAR-10 , dog and cat tend to confuse even a state-of-the-art DNN model since they share many common semantic features. As a result, the two classes tend to be very close with each other in the representation space and the decision boundary between them might not

(a) given laptop, a mouse is predicted (b) a surfing woman is misclassied as man

Figure 6.1: **Examples of confusion and bias errors found in [78]**

be "fine-grained" enough for correct classification on some dog can cat images. We denote the error-inducing classes as target classes. To fix the errors of the target classes, the model needs to take more effort to learn from them. For large and complex DNN models, complete training from scratch may not be possible. Sometimes no extra data can be collected, either. In these cases, fine-tuning can be applied to let the model to take more effort learning from target classes. When fine-tuning is not possible or training data cannot be accessed, (for example, the user does not have right to access the data) the output can be modified to fix the errors while sacrificing the overall performance to some extent.

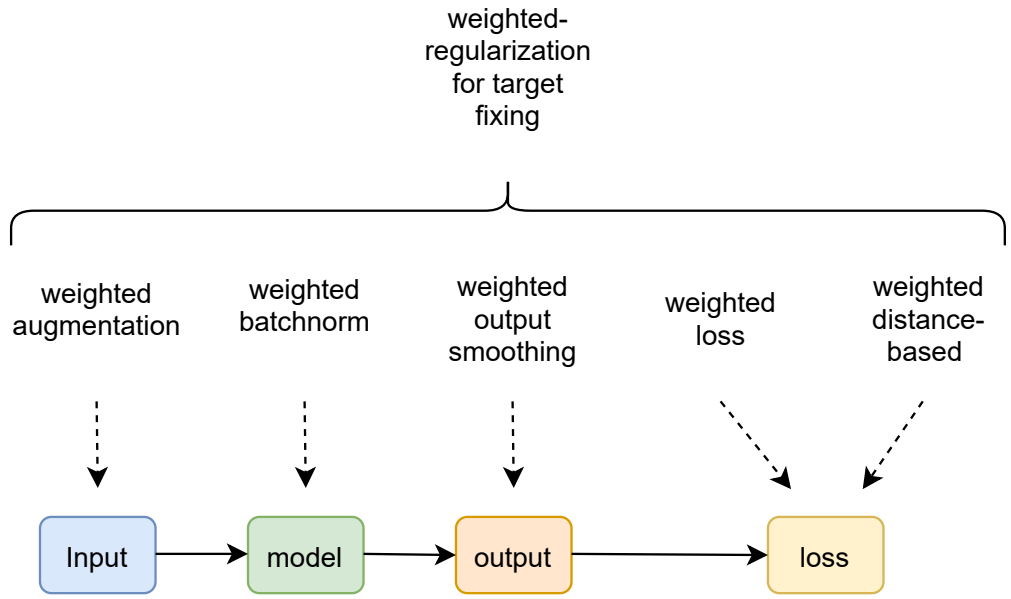


Figure 6.2: **Overview of Weighted Regularization for Target Fixing**

With this insight, we propose a generic method called weighted regularization (WR). WR consists of multiple methods including weighted augmentation (w-aug), weighted batch normalization (w-bn), weighted output smoothing (w-os), weighted loss (w-loss), and weighted distance-based regularization (w-dbr). These methods function at different stage of a given DNN's training or inference time Figure 6.2. In particular, if retraining is allowed and training data are provided, w-aug assigns more weights to the target classes during the retraining, w-bn shifts the distribution of the activation values induced by the input at every batchnorm layer(assuming the model has batchnorm layers), and w-loss and w-dbr modify the loss function by assigning more weights to the erroneous instances and regularizing the class centroids in the representation space, respectively. Such fine-tuning strategies enable the model to emphasize more on the instances of the target classes and thus more likely to avoid the errors involving the target classes. If fine-tuning and training data are not provided, w-os multiplies the model's prediction on target classes by a smaller user-specified constant. In other words, making the model predict less the target class. In this way, the group-level errors on those unsure data points can be avoided.

Figure 6.3 further illustrates the different impact of these methods using an example consists of three classes (square, circle and diamond). The colors represent the model's prediction while the dark dashed lines denote the model's decision boundary. Figure 6.3(a) shows that the original model tends to confuse between blue and red since these two classes are very close with each other. Ideally, a fixing method wants to finetune the model such that the decision boundary becomes that in Figure 6.3(d). w-os tends to solve the confusion issue by contracting the decision boundary of the target classes as illustrated in Figure 6.3(c). w-aug, w-loss, and w-dbr try to reuce confusion by shifting the decision boundary. They may be able to achieve Figure 6.3(d) but may also sacrifice the decision boundary for other classes sometimes and get the decision boundary in Figure 6.3(b) instead. w-bn comes in between: on the one hand, it tends to contract the decision boundary as w-os; on the other hand, it tends to shift the decision boundary through finetuning.

We evaluate the proposed methods on fixing confusion error and bias error for both single-label and multi-label image classification in five different settings involving four datasets and DNN

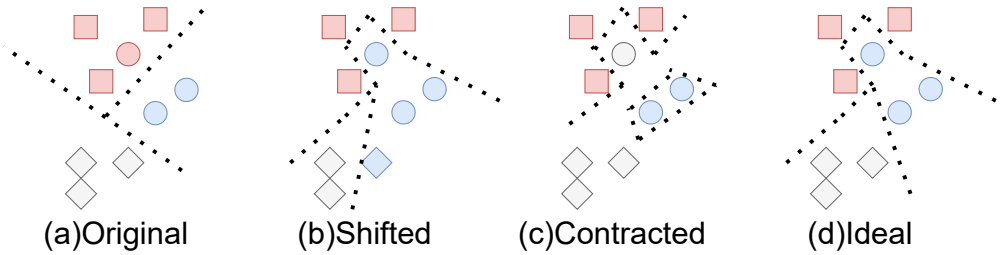


Figure 6.3: **Illustration of different potential decision boundary before and after applying WR.**

architectures. Our experiments show that WR can effectively fixing the errors with minimal cost of the overall performance. In every setting, a subset of our proposed methods can reduce the error significantly and most of time at least one method can achieve better accuracy (or mean average precision for multi-label image classification) and lower confusion/bias error than the original model at the same time. We also provide some analysis of the proposed methods performance and applicability. In summary, we make the following contributions:

- We propose a generic method for target class-level bug fixing of DNN models, WR.
- We compare the proposed, specific fixing methods under the generic method and show their effectiveness on fixing two types of class-level errors.

Our code is available at `https://github.com/deepfixdeepfix/dnnfix`.

6.2 Methodology

Figure 6.2 provides an overview of our proposed framework, Weighted Regularization (WR) for DNN target fixing. It consists of different methods applied to different stages of a DNN's retraining or inference. In particular, Weighted Augmentation (w-aug) reweights the input data according to their class membership, Weighted Batch Normalization (w-bn) modifies the batch norm layer's statistics of the given DNN model, Weighted Output Smoothing (w-os) smooths a model's prediction probability of each class, and finally, Weighted Loss (w-loss) and Weighted Distance-Based Regularization (w-dbr) assign more weights to a model's mistake on target classes and regularize the representation of the target classes in the loss function when finetuning the DNN model, respectively.

When training data is accessible and finetuning of the given model is allowed, w-aug, w-bn, w-loss, w-dbr can be applied to repairing the model through finetuning. w-os, on the other hand, does not require the access to the training data or the extra finetuning step. It only required the DNN's prediction confidence values for every class (mostly last layer's output). An extra limitation for w-bn is that it requires the given model to have batch norm layers.

In the following subsections, we will introduce the details of each method in terms of how they are developed to fix the confusion error and the bias error for both single-label classification as well as multi-label classification. For simplicity, we only explain our methods in fixing confusion error between one pair of classes and bias error among one triplet. However, our method can be very easily extended to fixing multiple pairs and multiple triplets by simply treating every pair / triplet the same way as the one demonstrated in the methodology. In the result section, we further show the effectiveness of applying our methods to fixing multiple confused pairs.

6.2.1 Original model (orig)

Before explaining our proposed methods, we first briefly introduce how the original model is trained. The original model is trained using the standard objective function as follows:

$$Loss_{orig} = \mathbb{E}_{(x,y)\sim\mathbb{D}}\mathbf{L}(f(x), y)$$

where $\mathbb{D}$ is the underlying data distribution of input x and label y and $\mathbf{L}$ is the user specified loss function. Some widely used classification loss functions include cross-entropy loss and l2 loss.

6.2.2 Weighted augmentation (w-aug)

The weighted augmentation method oversamples images from the target classes such that the DNN model will be able to better identify these target classes. For fixing the confusion error, the target classes consists of the chosen pairs of classes. For fixing the bias error, the target classes

consists of the chosen triplets of classes. The loss function is defined as:

$$Loss_{os} = \mathbb{E}_{(x,y)\sim\mathbb{D}'}\mathbf{L}(f(x), y)$$

where the probability density function for the weighted distribution $\mathbb{D}'$ is

$$pdf'(X,Y) = \begin{cases} \rho \times pdf(X,Y), & \text{if } y \in Y_{target} \\ pdf(X,Y), & \text{otherwise} \end{cases}$$

where pdf is the probability density function of the original data distribution D. In essence, the images that have labels belonging to the target classes are oversampled by a user specified constant $\rho \geq 1$ times during finetuning. The larger ρ is, more effort the DNN model will spend on the target classes compared with other non-target classes.

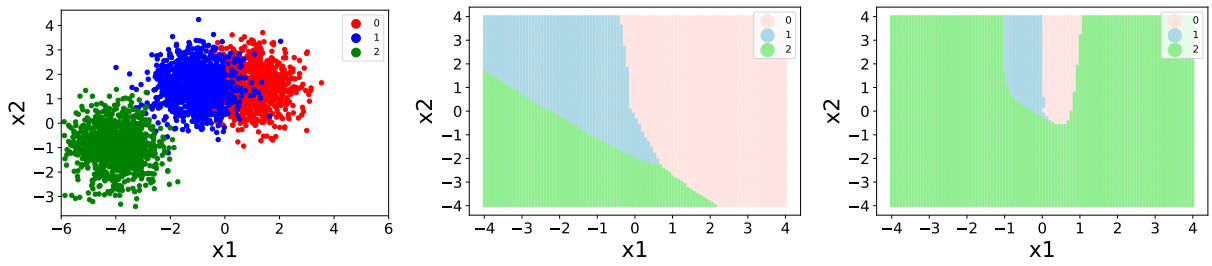
(a)Fixing confusion error: target classes A and B are oversampled.

(b)Fixing bias error: target classes A, B and C are oversampled.

6.2.3 Weighted batch normalization (w-bn)

The weighted batch normalization method redistributes the batch normalization layer based on the distribution of data from target classes. Such method has not been proposed before in previous literature. We have found that this method can shift the decision boundaries of the non-target classes towards the target classes. To demonstrate this phenomenon, Figure 6.4(a) shows a toy 2D dataset composed of three classes. Figure 6.4(b) shows the decision boundary of a well-trained simple ResNet model and Figure 6.4(c) shows the decision boundary of the model retrained via w-bn. It is noticeable that the decision boundary of class 2 expand over class 0 and class 1.

Figure 6.5 illustrates our approach. We first denote x to be a regular batch of images and x^{target} to be a batch consists of images sampled only from the target classes. (a) shows a traditional BN



(a) toy 2D dataset (b) original model (c) weighted BN

Figure 6.4: **Shift of decision boundary using weighted BN**

layer. Given the input from previous convolutional layer $x_{(1)}$, the output

$$x_{(2)} = \frac{x_{(1)} - E[x_{(1)}]}{\sqrt{Var[x_{(1)}] + \epsilon}} \times \gamma + \beta.$$

where γ and β are standard batchnorm scaling and shifting trainable parameters.

In contrast, the proposed weighted BN layer (shown in (b)) takes in both $x_{(1)}$ and $x_{(1)}^{target}$ and has an output

$$x_{(2)} = \frac{x_{(1)} - \widehat{E}[x_{(1)}, x_{(1)}^{target}]}{\sqrt{\widehat{Var}[x_{(1)}, x_{(1)}^{target}] + \epsilon}} \times \gamma + \beta$$

where

$$\widehat{E}[x_{(1)}, x_{(1)}^{target}] := (1-\rho)E[x_{(1)}] + \rho E[x_{(1)}^{target}],$$

$$\widehat{Var}[x_{(1)}, x_{(1)}^{target}] := (1-\rho)Var[x_{(1)}] + \rho Var[x_{(1)}^{target}].$$

The main differences are that the weighted BN layer passes extra batch of the target classes and assigns more weights to those data when estimating the BN statistics (i.e. batch mean E and batch variance Var) in neural network forward pass. It should be noted that during the back-propagation, only the loss coming from the regular batch (highlighted in dashed red box) is considered. This is because we aim to preserve the overall accuracy at the same time of reducing the confusion/bias error.

(a)Fixing confusion error: More instances of the uncertain target classes A and B (those lay around the decision boundary) tend to be predicted to other classes. Consequently, the confusion between

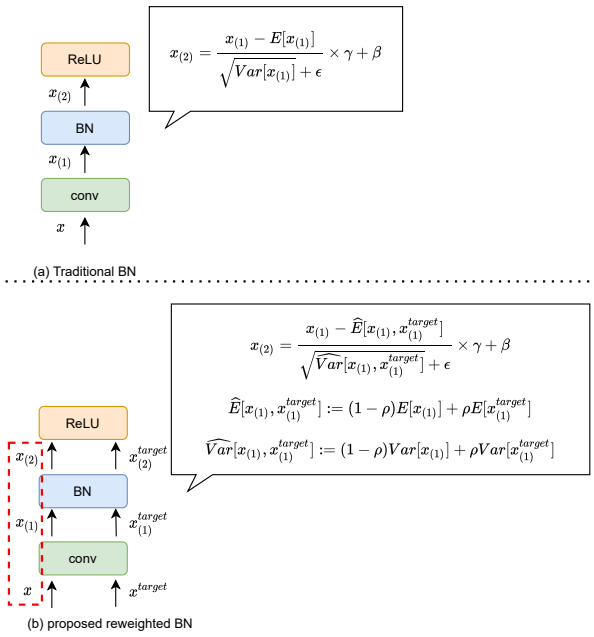


Figure 6.5: **Illustration of the traditional BN layer and the proposed weighted BN layer.**

class A and B drops while sacrificing the confusion between class A, B and other classes.

(b)Fixing bias error: all the three classes A, B, and C will be included in the target class. The reason is that this will contract the decision boundaries of all the three classes and the misclassified ones among them will be more likely to be predicted to other classes. As a result, both confusions between A and C, and B and C tend to drop. It follows that the bias will be mitigated.

6.2.4 Weighted output smoothing (w-os)

First we denote the last layer's output of a given input x to be $p(x)$, which is a m (i.e. the number of classes) dimensional vector. Each field of $p(x)$ is positively correlated with the prediction probability of the class corresponding to that field.

(a)Fixing confusion error: for single-label classification, w-os multiplies the target class prediction probability $p(x)_t$ by a specified parameter $\eta \in [0,1]$ for images classified into any of the target classes A and B; for multi-label classification, w-os multiplies the target class prediction probability $p(x)_t$ by η for images predicted to have both target classes A and B.

(b)Fixing bias error: for single-label classification, w-os multiplies the target classes prediction probability $p(x)_t$ by η for images predicted to be any of the target classes A, B, and C; for multi-

label classification, w-os multiplies the target classes prediction probability $p(x)_t$ by η for images predicted to have the positive target class A and the anchor target class C or have the negative target class B and the anchor target class C.

In essence, this method tends to make those unsure prediction on the target classes to be predicted to the most confident non-target classes. Thus, images laying around the decision boundary of the target classes tend to change prediction early and thus reduce the confusion between target classes. In order to repair bias error, it reduces the confusion between the two pairs at the same time. Since most of the influence will be on the higher confused pair(for example A-B), it does not hurt the overall accuracy much.

Note that this method is analogous to the post-processing method in fairness ML literature[264], where different thresholds are assigned to each sensitive group for a binary classifier's to mitigate equalized odds/opportunity. The thresholds are set to make trade-off between prediction accuracy and fairness criteria. In contrast, we apply different levels of smoothing for target classes to make trade-off between accuracy and confusion/bias errors.

6.2.5 Weighted loss (w-loss)

The weighted loss method allows a user to assign more weights to images leading to a confusion error or a bias error.

(a)Fixing confusion error: Denote $Y_{target} = \{A, B\}$. The loss function is defined as:

$$Loss_{rl} = (1-\rho)\mathbb{E}_{(x,y)\sim\mathbb{D}}\mathbf{L}(f(x), y)$$
$$+ \rho\mathbb{E}_{(x,y)\sim\mathbb{D}(Y_{target})}\mathbf{L}(f(x), y)$$

where the probability density function for the distribution $\mathbb{D}(Y_{target})$ is

$$pdf'(X,Y) = \begin{cases} pdf(X,Y), & \text{if } (x,y) \sim \mathbb{D} \text{ s.t. } y \in Y_{target} \\ & \text{and } f(x) \neq y \text{ and } f(x) \in Y_{target}. \\ 0, & \text{otherwise} \end{cases}$$

Intuitively, the DNN model is encouraged to better differentiate between A and B compared with differentiating among other classes in general.

(b)Fixing bias error: The loss function is defined as:

$$\begin{aligned} Loss_{rl} = (1-\rho)\mathbb{E}_{(x,y)\sim\mathbb{D}}\mathbf{L}(f(x),y) \\ + \rho\Big(\mathbb{E}_{(x,y)\sim\mathbb{D}'(Y_{target^+})}\mathbf{L}(f(x),y) \\ + \mathbb{E}_{(x,y)\sim\mathbb{D}(Y_{target^-})}\mathbf{L}(f(x),y)\Big) \end{aligned}$$

where $Y_{target^+} = \{A, C\}$ and $Y_{target^-} = \{B, C\}$. This loss function encourages the DNN model to better differentiate between A and C as well as B and C compared with differentiating among other classes in general.

6.2.6 Weighted distance-based regularization (w-dbr)

The distance-based regularization method leverages the class-level representation and adds an extra regularization term in the loss function to balance the distance among the target classes in the representation space under a defined metric. The insight here is that the closer the two classes representations are, the more confused the model is between the two classes[78].

Given a class A, we define its class-level representation

$$P_{new}(A) = \frac{[S(n_1), S(n_2), ..., S(n_t)]}{N},$$

where $S(n_i)$ is the sum of each output of neuron n_i, given N input images. Then, we define the

distance metric between two classes A and B as:

$$D_{new}(A, B) = ||(P_{new}(A), P_{new}(B))||_2$$

(a)Fixing confusion error: The loss for reducing confusion is defined as:

$$Loss_{dbr-conf} = Loss_{orig} - \rho D_{new}(A, B)$$

where ρ trades off the original loss and the new distance-based regularization. In essence, the regularization term encourages a larger separation of the centroids of the two classes A and B in the representation space.

(b)Fixing bias error: Similarly, we define a new loss for reducing bias:

$$Loss_{dbr-bias} = Loss_{orig} + \rho\ abs(D_{new}(A, C) - D_{new}(B, C)).$$

The regularization term balances the difference between the centroid distance between class A and C, and centroid distance between class B and C in the representation space such that the relative distances from A to B and C are similar.

6.3 Experimental Design

6.3.1 Study Subjects

We evaluate the proposed method for two applications involving both single-label and multi-label DNN-based classifications including five combinations of four DNN architectures for four datasets.

Datasets: We conduct our experiments on two single-label image classification datasets, CIFAR-10, CIFAR-100 and two multi-label image classification datasets, MS-COCO[105] and MS-COCO gender[83].

- **CIFAR-10**: consists of 50,000 training and 10,000 testing 32x32 color images. It has 10 classes and 6,000 images per class.
- **CIFAR-100**: consists of 50,000 training and 10,000 testing 32x32 color images. It has 100 classes and 600 images per class.
- **MS-COCO**: MS-COCO dataset has 80 objects. It contains 80783 training images and 40504 validation images.
- **MS-COCO gender**: MS-COCO gender dataset is a subset of MS-COCO dataset. The gender information is annotated by Zhao etal[83]. We remove person class and add man and woman classes based on the gender annotation.

Architectures: We evaluate our repairing performance on four different convolutional neural networks[209, 211].

- **ResNet-18**: ResNet-18 is trained on CIFAR-10 dataset. the model is trained using the state-of-the-art training scripts from CutMix[265]. The training takes 300 epochs and the repairing takes 60 epochs for methods requireing retraining. The initial learning rate is 0.1 and is multiplied by 0.1 after 50% training epochs and 75% training epochs.
- **VGG11_BN**: VGG11_BN is a variant of VGG11 model with batch normalization layers [211]. We train a VGG11_BN model on CIFAR-10 dataset in a same way as above.
- **ResNet-34**: ResNet-34 is trained on CIFAR-100 dataset. the model is trained in the same way as above.
- **ResNet-50**: Following Zhao et al[83], we train ResNet-50 models for both MS-COCO and MS-COCO gender datasets. Both models are trained for 12 epochs and are repaired by retraining of another 6 epochs for methods requiring retraining.

Table 6.1 summarizes our study subjects including the details of all the datasets and models used.

Table 6.1: **Study Subjects**

Dataset			Model			
Classification Task	**Name**	**#classes**	**Models**	**#Params**	**#Layers**	**Reported Accuracy**
Multi-label classification	COCO [105]	80	ResNet-50[209]	23,671,952	174	0.6603*
	COCO gender[83]	81	ResNet-50[209]	23,674,001	174	0.6691*
Single-label classification	CIFAR-100[103]	100	ResNet-34 [265]	336,244	101	0.6961†
	CIFAR-10[103]	10	ResNet-18[265]	127,642	41	0.8747†
			VGG11-BN[211]	9,756,426	36	0.9175†

* reported in mean average precision, †reported in mean accuracy

6.3.2 Evaluations Metrics

For either fixing the confusion error or bias error, the goal is to fix the error while maintaining or even improving the model's overall accuracy. Since there are two goals i.e. high accuracy and low confusion/bias a model tries to achieve, for comparison purpose, we rank each fixed model (including the original model) by accuracy and confusion respectively. Next, we sum up the two ranks for each model and compare the rank sums. The model with the smallest rank sum is considered the one that achieves the best trade-off between accuracy and confusion/bias.

Research Questions. With the experimental setting mentioned above, we investigate the following two research questions to evaluate WR for target bug fixing of DNNs:

- **RQ1.** Can WR fixes confusion errors of DNN models for both single-label classification and multi-label classification effectively?
- **RQ2.** Can WR fixes bias errors of DNN models for both single-label classification and multi-label classification effectively?

6.4 Results

RQ1. Fixing Confusion Error

In this RQ, we explore if the proposed methods can fix confusion errors effectively. We evaluate the proposed methods on two settings in the multi-label classification task and three settings in the

single-label classification task. We choose two hyper-parameters for each method and thus result in eleven models (including the original model) for each setting.

Table 6.2 shows the results under different settings. We highlight the top3 (or top4 if tied) methods in terms of rank sum. In summary, under every setting, many of the proposed methods can achieve lower confusion while preserving decent overall accuracy (or mean average precision for multi-label classification). For example, w-os can almost always decrease the confusion to close to 0 under every setting while maintaining accuracy at a reasonable level.

For the multi-label classification task, w-loss strikes the best trade-off between mean average precision and confusion in terms of the rank sum. On both the COCO and COCO gender datasets, w-loss improves the mean average precision while also decreasing confusion compared with the original model. For example, on the COCO dataset, w-loss(0.4) improves the mean average precision from 0.6603 to 0.6611 and reduces confusion between person and bus from 0.2381 to 0.03457; on the COCO gender dataset, w-loss(0.4) improves the mean average precision from 0.6691 to 0.6697 and reduces confusion between handbag and woman from 0.0394 to 0.02063.

For the single-label classification task, on CIFAR-10 , w-dbr(0.1) is ranked the top while on CIFAR-100 , w-aug(5) is ranked the top. They also achieve higher accuracy and lower confusion compared with the original models in the same setting, respectively. In particular, on the CIFAR-10 dataset and ResNet-18 model, w-dbr(0.1) improves the overall accuracy from 0.8747 to 0.8764 and reduces confusion between cat and dog from 0.096 to 0.09; on the CIFAR-10 dataset and VGG-11 with BN model, w-bn(0.4) improves the overall accuracy from 0.9175 to 0.919 and reduces confusion between cat and dog from 0.083 to 0.076; on the CIFAR-100 dataset, w-aug(5) improves the accuracy from 0.6961 to 0.6697 and reduces confusion between girl and woman from 0.15 to 0.12.

Under all the settings, w-os works very well and gives decent trade-off between accuracy and confusion. In particular, w-os with different hyper-parameters is ranked at the top3 for every setting. On the flip side, it has to trade confusion for accuracy. w-bn also works reasonably well across all the settings but slighly worse than w-os in most settings. Although w-loss works

Table 6.2: **Results on Confusion**

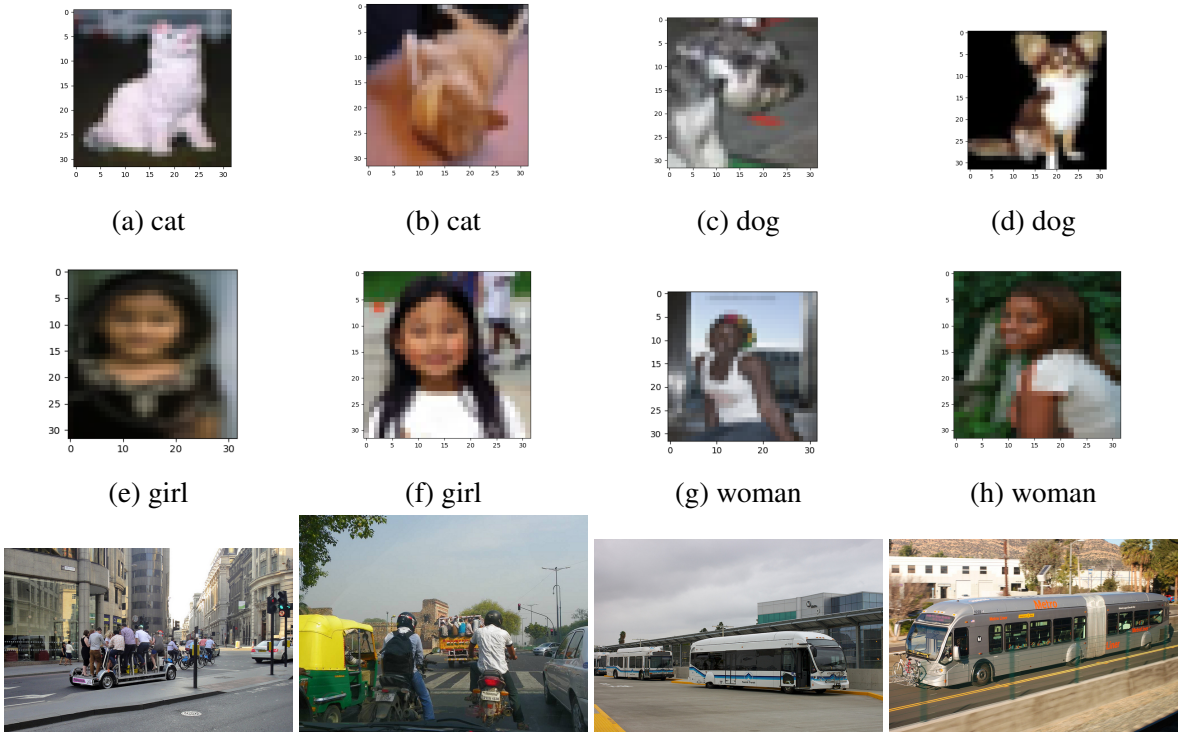
Dataset	Model	Target Classes	Method	Accuracy	Confusion	Acc Rank	Conf Rank	Rank Sum
COCO *	ResNet-50	person, bus	orig	0.6603	0.2381	4	9	13
			w-aug(3)	0.6563	0.2763	10	10	20
			w-aug(5)	0.6531	0.3009	11	11	22
			w-bn(0.4)	0.6606	0.101	2	7	9
			w-bn(0.6)	0.6585	0.0641	8	6	14
			w-os(0.5)	0.6599	0	6	1	7
			w-os(0.7)	0.6602	0.1159	5	8	13
			w-loss(0.4)	0.6606	0.03457	2	4	6
			w-loss(0.6)	0.6611	0.04276	1	5	6
			w-dbr(0.5)	0.6589	0.01309	7	2	9
			w-dbr(1)	0.6566	0.02491	9	3	12
COCO gender*	ResNet-50	handbag, woman	orig	0.6691	0.0394	3	7	10
			w-aug(3)	0.6679	0.0643	7	8	15
			w-aug(5)	0.6646	0.0897	11	11	22
			w-bn(0.4)	0.6689	0.00603	4	3	7
			w-bn(0.6)	0.6679	0	7	1	8
			w-os(0.5)	0.6686	0	6	1	7
			w-os(0.7)	0.6688	0.0061	5	4	9
			w-loss(0.4)	0.6697	0.02063	1	6	7
			w-loss(0.6)	0.6695	0.01754	2	5	7
			w-dbr(0.5)	0.6672	0.06763	9	9	18
			w-dbr(1)	0.6659	0.06867	10	10	20
CIFAR-100	ResNet-34	girl, woman	orig	0.6961	0.15	3	8	11
			w-aug(3)	0.7042	0.145	2	7	9
			w-aug(5)	0.7043	0.12	1	5	6
			w-bn(0.4)	0.6593	0.02	7	3	10
			w-bn(0.6)	0.6174	0.005	8	1	9
			w-os(0.01)	0.6911	0.055	4	4	8
			w-os(0.001)	0.6901	0.01	5	2	7
			w-loss(0.4)	0.534	0.17	10	9	19
			w-loss(0.6)	0.5342	0.165	9	8	17
			w-dbr(0.1)	0.6628	0.125	6	6	12
			w-dbr(0.5)	0.6593	0.145	7	7	14
CIFAR-10	ResNet-18	cat, dog	orig	0.8747	0.0960	3	9	12
			w-aug(3)	0.875	0.093	2	8	10
			w-aug(5)	0.8636	0.0915	6	7	13
			w-bn(0.4)	0.8313	0.051	8	3	11
			w-bn(0.6)	0.781	0.028	10	2	12
			w-os(0.1)	0.8654	0.0710	5	4	9
			w-os(0.001)	0.8056	0.0195	9	1	10
			w-loss(0.4)	0.8566	0.1105	7	10	17
			w-loss(0.6)	0.7556	0.172	11	11	22
			w-dbr(0.1)	0.8764	0.09	1	6	7
			w-dbr(0.5)	0.8699	0.087	4	5	9
	VGG-11 with BN	cat, dog	orig	0.9175	0.083	2	9	11
			w-aug(3)	0.9143	0.0655	6	2	8
			w-aug(5)	0.9138	0.07	8	3	11
			w-bn(0.4)	0.919	0.076	1	6	7
			w-bn(0.6)	0.9167	0.0725	4	4	8
			w-os(0.1)	0.9173	0.081	3	8	11
			w-os(0.001)	0.9086	0.053	9	1	10
			w-loss(0.4)	0.7912	0.3135	10	10	20
			w-loss(0.6)	0.2017	0.4125	11	11	22
			w-dbr(0.1)	0.9151	0.076	5	6	11
			w-dbr(0.5)	0.9141	0.0725	7	4	11

* reported in mean average precision

very well on COCO and COCO gender, it works poorly on CIFAR-10 and CIFAR-100 . A deeper exploration of the retraining process reveals that its retraining processes on CIFAR-10 and CIFAR-100 tend to be very unstable. For example, on CIFAR-100 , it tends to misclassify dog to cat much more frequently on one epoch and the reverse on another. w-aug is ranked among the top on CIFAR-100 and CIFAR-10 but performs quite poorly on COCO and COCO gender. This is because the causes of confusions between single-label classification and multi-label classification are different. For CIFAR-100 and CIFAR-10 , the high confusions result from similar features between two classes when the target classes are oversampled during training, the difference of features are better learned by models. However, for COCO and COCO gender the confusions mostly result from two objects appearing together frequently in the same image or having similar backgrounds. When target classes are oversampled, the confusions may be increased. w-dbr can reduce confusion for COCO , CIFAR-100 and CIFAR-10 but fail to do so on COCO gender. One possibility is that the confusion between handbag and woman is already very low and the class centroids between woman and handbag are far away from each other so the extra loss regularization term does not help reduce the confusion further.

Figure 6.6 shows some examples of the fixed confusion instances. On the CIFAR-10 dataset, Figure 6.6(a)-(b) show two cat images that were originally classified to dog by the original model. After the fixing using w-dbr, they are correctly predicted to cat and Figure 6.6(c)-(d) show two dog images that were originally classified to cat by the original model. After the fixing using w-dbr, they are correctly predicted to dog after applying w-dbr. Similarly, on the CIFAR-100 dataset Figure 6.6(e)-(f) show two girl images that were originally classified to woman by the original model. After the fixing using w-dbr, they are correctly predicted to girl. Figure 6.6(g)-(h) show two woman images that were originally classified to girl by the original model. After the fixing with w-dbr, they are correctly predicted to woman. On the COCO dataset, Figure 6.6(e)-(f) show two images that contain only person but the original model mispredicts the presence of bus in them. After fixing using w-loss, the model correctly predicts the presence of person without false positively predicting the presence of bus. Similarly, Figure 6.6(g)-(h) show two images with only

bus in them but the original model false positively predicts the presence of person in them as well. After fixing using w-loss, the model can correctly predict the presence of bus while not falsely predicting the presence of person in the images.



(a) cat (b) cat (c) dog (d) dog

(e) girl (f) girl (g) woman (h) woman

(i) person (j) person (k) bus (l) bus

Figure 6.6: **Fixed confusion errors on CIFAR-10 ((a)-(d)), CIFAR-100 ((e)-(h)), and COCO ((i)-(l)) respectively.**

Next, we explore if the proposed methods can be applied to fixing confusion errors among multiple pairs at the same time. To answer this question, we apply the proposed methods to fixing confusions between two pairs (one top confused pair and one randomly picked confused pair) on CIFAR-10 and COCO . Table 6.3 shows the results. On COCO , w-loss(0.6) achieves the best trade-off between accuracy and confusion. It reduces the summed confusions of two pairs (person-bus and mouse-keyboard) from 0.4025 to 0.0723 while only sacrificing mean average precision by 0.0001 (from 0.6604 to 0.6603). On CIFAR-10 , w-dbr(0.1) achieves the best trade-off by increasing the overall accuracy from 0.8747 to 0.8778 and reducing the confusion from 0.134 to 0.128. We also want to highlight that w-os is ranked in top3 for both settings and significantly

reduces the confusion in both settings while only slightly sacrificing the mean average precision and accuracy, respectively.

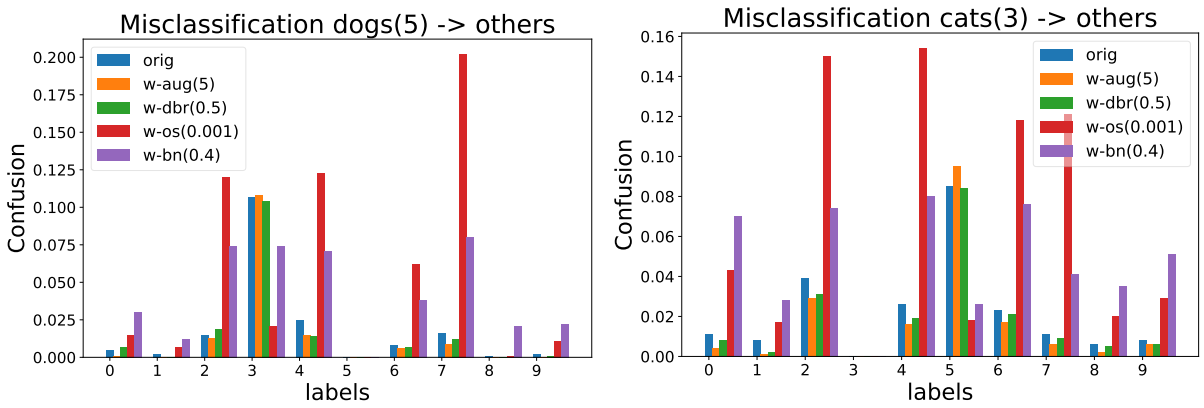
Table 6.3: **Results on Confusion (Two Pairs)**

Dataset	Model	Target Classes	Method	Accuracy	Confusion	Acc Rank	Conf Rank	Rank Sum
COCO *	ResNet-50	(person, bus), (mouse, keyboard)	orig	0.6604	0.4025	1	6	7
			w-aug(3)	0.6564	0.5335	7	7	14
			w-bn(0.6)	0.6574	0.2032	5	4	9
			w-os(0.7)	0.6595	0.1937	3	3	6
			w-os(0.5)	0.6575	0	4	1	5
			w-loss(0.6)	0.6603	0.0723	2	2	4
			w-dbr(0.5)	0.6566	0.2954	6	5	11
CIFAR-10	ResNet-18	(cat, dog), (automobile, truck)	orig	0.8747	0.134	2	5	7
			w-aug(3)	0.8697	0.149	3	6	9
			w-bn(0.6)	0.8328	0.082	6	2	8
			w-os(0.1)	0.8628	0.1055	4	3	7
			w-os(0.001)	0.7788	0.0327	7	1	8
			w-loss(0.4)	0.8493	0.1655	5	7	12
			w-dbr(0.1)	0.8778	0.128	1	4	5

* reported in mean average precision

We further check the confusion between the target classes and all other classes on CIFAR-10 to explore how the confusion from the target classes to other classes change when using different methods. Figure 6.7 shows the result. In the original model, dog(label 5) is highly confused with cat(label 3) than any other classes. After the fixing, the confusion between dog and cat drops slightly for w-aug and w-dbr and drops significantly for w-os and w-bn. However, the confusion between dog and other classes (also cat and other classes) increase at the same time as a trade-off for all the methods except w-aug. It is also worth noting that w-bn(the purple bars) provides a uniform distribution of the confusion after fixing. This might be a desirable property if a user does not want to overburden one particular non-target class in terms of the confusion distribution. The result also suggests a future exploration direction of our method: by adjusting hyper-parameter, one can optimize the model such that the maximum pair-wise confusion is the lowest. In other words, no pair should be confused much larger than other pairs.

Since w-os can be applied when no training data is available or retraining is allowed and can reduce confusion by a significant amount while only slightly sacrificing the overall performance under every setting, we conduct an ablation study on its hyper-parameter η to explore its trade-off between confusion and accuracy. Figure 6.8 shows the results. Note that by decreasing η,



(a) confusion between dog and other classes (b) confusion between cat and other classes

Figure 6.7: **Confusion between target classes and non-target classes for each model on CIFAR-10 .**

the confusion decreases and accuracy decrease at the same time. Thus, a user can decide what parameter to use depending on the significance of accuracy and confusion.

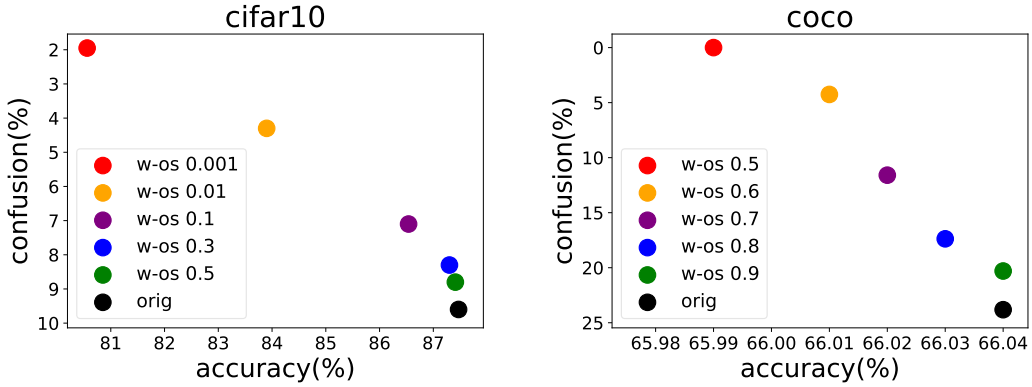


Figure 6.8: **Accuracy and Confusion trade-off of different parameters for w-os.**

> **Result 1:** *The proposed method WR can effectively fix confusion errors for both single-label image classification and multi-label image classification. Under every setting, one fixing method can achieve both higher accuracy and lower confusion than the original model. The proposed methods also generalize to fixing errors for multiple pairs.*

RQ2. Fixing Bias Error

In this RQ, we explore if the proposed methods can fix bias errors. The settings are similar to those for evaluating confusion error fixing. Table 6.4 shows the results under different settings. In

summary, the general trend is similar to fixing the confusion error. Under every setting, many of the proposed methods can achieve lower bias while preserving decent overall accuracy (or mean average precision for multi-label classification). For example, w-os can significantly decrease the bias under every setting.

For the multi-label classification task, w-loss strikes the best trade-off between mean average precision and bias in terms of the rank sum. On both the COCO and COCO gender datasets, w-loss improves the mean average precision while also decreasing the bias compared with the original model. For example, on the COCO dataset, w-loss(0.6) improves the mean average precision from 0.6603 to 0.6611 and reduces bias between person and clock with respect to bus from 0.2366 to 0.0425; on the COCO gender dataset, w-loss(0.4) improves the mean average precision from 0.6691 to 0.6706 and reduces bias between woman and man with respect to skis from 0.2630 to 0.02472.

For the single-label classification task, on CIFAR-10 and ResNet-18 model, w-dbr(0.5) is ranked among the top3 while on CIFAR-100 , w-aug(5) is ranked among the top3. Both methods achieve higher accuracy and lower bias compared with the original models in the same setting, respectively. In particular, on CIFAR-10 and ResNet-18 model, w-dbr(0.1) improves the overall accuracy from 0.8747 to 0.8763 and reduces the bias between dog and cat with respect to bird from 0.092 to 0.062; on CIFAR-100 , w-aug(5) improves the accuracy from 0.6961 to 0.7059 and reduces the bias between girl and boy with respect to woman from 0.09 to 0.075. On CIFAR-10 and VGG-11 with BN model, w-aug(3) has the top1 performance. It significantly decreases the bias from 0.065 to 0.04 with only slightly decreasing of accuracy from 0.9175 to 0.914.

Similar to fixing confusion errors, under all the settings, w-os works reasonably well and gives decent trade-off between accuracy and bias. The observations for w-bn, w-loss, w-aug, and w-dbr are also similar as in performance in fixing confusion errors. In particular, w-bn also can reduce bias significantly in most settings but tends to be slightly worse than w-os overall. w-loss works very well for the multi-label classification task but not for the single-label classification task. w-aug performs much better on CIFAR-100 and CIFAR-10 than on COCO and COCO gender. w-dbr

Table 6.4: **Results on Bias**

Dataset	Model	Target Classes	Method	Accuracy	Bias	Acc Rank	Conf Rank	Rank Sum
COCO *	ResNet-50	bus, person, clock	orig	0.6603	0.2366	3	6	9
			w-aug(3)	0.6563	0.2760	10	8	18
			w-aug(5)	0.6532	0.2951	11	9	20
			w-bn(0.4)	0.6581	0.2445	8	7	15
			w-bn(0.6)	0.6576	0.1983	9	5	14
			w-os(0.5)	0.6599	0	5	1	6
			w-os(0.7)	0.6602	0.1160	4	4	8
			w-loss(0.4)	0.6607	0.04131	2	2	4
			w-loss(0.6)	0.6611	0.0425	1	3	4
			w-dbr(0.5)	0.6595	0.3589	7	11	18
			w-dbr(1)	0.6599	0.3557	5	10	15
COCO gender*	ResNet-50	skis, woman, man	orig	0.6691	0.2630	3	8	11
			w-aug(3)	0.6648	0.2972	9	10	19
			w-aug(5)	0.6623	0.3190	11	11	22
			w-bn(0.4)	0.6656	0.1277	8	5	13
			w-bn(0.6)	0.6645	0.0861	10	4	14
			w-os(0.5)	0.6685	0	6	1	7
			w-os(0.7)	0.6689	0.2119	4	6	10
			w-loss(0.4)	0.6706	0.02472	1	3	4
			w-loss(0.6)	0.6703	0.02232	2	2	4
			w-dbr(0.5)	0.6687	0.2606	5	7	12
			w-dbr(1)	0.6684	0.2877	7	9	16
CIFAR-100	ResNet-34	woman, girl, boy	orig	0.6961	0.09	3	11	14
			w-aug(3)	0.7002	0.075	2	9	11
			w-aug(5)	0.7059	0.075	1	9	10
			w-bn(0.2)	0.6892	0.040	5	4	9
			w-bn(0.4)	0.6584	0.0050	10	1	11
			w-os(0.001)	0.688	0.01	6	2	8
			w-os(0.1)	0.6944	0.06	4	6	10
			w-loss(0.1)	0.6748	0.06	7	6	13
			w-loss(0.4)	0.5804	0.07	11	8	19
			w-dbr(0.1)	0.6668	0.04	8	4	12
			w-dbr(0.5)	0.6639	0.015	9	3	12
CIFAR-10	ResNet-18	dog, cat, bird	orig	0.8747	0.092	2	10	12
			w-aug(3)	0.8696	0.073	4	8	12
			w-aug(5)	0.8605	0.0705	5	7	12
			w-bn(0.4)	0.8498	0.053	6	4	10
			w-bn(0.6)	0.8103	0.047	7	3	10
			w-os(0.01)	0.7675	0.016	10	1	11
			w-os(0.5)	0.8731	0.04	3	2	5
			w-loss(0.4)	0.7964	0.0915	8	9	17
			w-loss(0.6)	0.7348	0.1005	11	11	22
			w-dbr(0.5)	0.8763	0.062	1	6	7
			w-dbr(1)	0.7845	0.060	9	5	14
	VGG-11 with BN	dog, cat, bird	orig	0.9175	0.065	1	9	10
			w-aug(3)	0.914	0.04	6	1	7
			w-aug(5)	0.9123	0.048	7	3	10
			w-bn(0.4)	0.915	0.057	5	7	12
			w-bn(0.6)	0.9161	0.0525	3	4	7
			w-os(0.1)	0.9166	0.063	2	8	10
			w-os(0.001)	0.9012	0.054	9	6	15
			w-loss(0.4)	0.8031	0.1780	10	11	21
			w-loss(0.6)	0.6395	0.1715	11	10	21
			w-dbr(0.5)	0.916	0.0445	4	2	6
			w-dbr(1)	0.9099	0.050	8	5	13

* reported in mean average precision

can reduce bias for COCO , CIFAR-100 and CIFAR-10 but fail to do so on COCO gender.

Figure 6.6 shows two examples of the fixed bias instances. In particular, Figure 6.9(a) shows an image containing a woman and a skis but the original model classifies the woman to man. After fixing using w-loss, the model correctly predicts the presence of woman and skis. Figure 6.9(b) shows an image containing several woman, man and skis while the original model only predicts the presence of only man and skis but misses woman. After fixing the model using w-loss, the model successfully recovers the presence woman, man and skis.

(a) woman, skis (b) woman, man, skis

Figure 6.9: **Fixed bias errors on COCO gender.**

Result 2: *The proposed method WR can effectively fix bias errors for both single-label image classification and multi-label image classification. Under every setting, one fixing method can achieve both higher accuracy and lower bias than the original model.*

6.5 Related Work

6.5.1 Software Repairing

Automatic software repairing is very challenging and most of existing automatic program repairing works focus on traditional software.[23]. Traditional automatic repairing techniques include random or guided mutation of AST(Abstract Syntax Tree)[24, 25, 26, 27, 28, 29, 30, 31],

static program analysis or symbolic execution/concrete execution[32, 33, 34, 35, 36, 37]. The most recent techniques involve language models training and program synthesis[38, 39]. All these techniques proposed to repair traditional programs such as C, C++, Java or Python, cannot work on DNN based software because there is no program logic or AST in DNN models. In this paper, we propose, compare and discuss Weighted Regularization for automatic target repairing techniques on DNN based software repair for group-level errors.

6.5.2 DNN Testing and Repairing

An increasing number of works in SE for AI area focus on DNN testing and repairing. The testing techniques usually leverage metamorphic relation as oracle and coverage guided image transformation or perturbation for generating test cases[44, 53, 50, 54, 196, 266, 56]. Data augmentation and retraining techniques are usually proposed for repairing DNN models in improving overall accuracy[63, 64, 65]. There are also works in improving robustness of models against adversarial instances[55, 60, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75]. All of these papers focus on repairing instance-wise bugs i.e. a model's failure on the variations of a given image. In contrast, our paper focuses on fixing group-level errors.

6.5.3 Fairness

Fairness is an important problem from both a theoretical and a practical perspective [134, 135, 136, 137]. Related works in fairness usually define a fairness criteria and then optimize the original objective while satisfying the fairness criteria [138, 139, 140, 141, 142, 143]. These properties are defined either at individual [138, 144, 145] or group levels [146, 139, 147]. Our paper focuses on the fixing techniques of a group level fairness definition called bias error proposed in [78], and to the best of our knowledge, our proposed repairing methods for which have not been studied in the previous literature.

6.6 Discussion & Threats to Validity

There are many potential ways to fix group-level errors of DNNs. To mitigate this threat, we propose and compare the performance of five different methods. For each method, we set a parameter to make trade-off between accuracy and confusion. We only rank the results using two parameters for each method to compare different methods' performance.

There are many datasets and models, which can be used for the evaluation purpose. We choose five combinations of four widely used datasets and models in the computer vision field. Besides, DNN training is stochastic so the results may have some fluctuations. The general patterns of the results we observe hold across different runs of the same methods. Lastly, our method can be applied to DNN models used in applications beyond image classifications such as object detection and recommendation systems. We leave that for future work.

6.7 Conclusion

In this work, we propose a generic method called Weighted Regularization(WR) that can fix group-level errors for deep neural networks. To the best of our knowledge, this is the first work proposing, exploring and comparing target fixing methods, which can be applied in different stages of DNN retraining or inference, on repairing group-level DNN model errors. Our experimental results show that WR can effectively fix confusion and bias errors and these methods all have their pros, cons and applicable scenarios.

Chapter 7: Conclusion

7.1 Summary of Contribution

In this thesis, we show that the traditional software engineering techniques do not work on DNN based software because of no control flow, data flow, or AST in DNN based software and discussed the necessity of developing new software engineering techniques for DNN based software to ensure the functionality, safety as well as fairness. To this end, new challenges such as test generation, software testing, software repairing, software fairness should be addressed for DNN based software. We firstly introduced the development of the new area SE for AI, an intersection area between software engineering and artificial intelligence and then presented four major projects addressing these new challenges in SE for AI area. Our first project (Chapter 3) designed and implemented systematic testing tool for DNN based software. We applied our tool on three top performing DNN based self-driving car models in Udacity self-driving car challenge and it identified thousands of erroneous behaviours. Our second project studied per-point robustness of DNNs under natural variation. We proposed both white-box and black-box approaches to identify non-robust data points for DNN based image classifiers and DNN based self-driving car models. We implemented and evaluated our approaches on 9 DNN based image classifiers and 3 DNN based self-driving car models. Our approaches can identify weak points with reasonably good precision and recall for both DNN based image classifiers and self-driving cars. Our third project(Chapter 5) proposed a neuron coverage based distance metric to identify group-level errors(confusion and bias errors) for DNN based software and it identified group-level errors in widely used models trained on popular single-label and multi-label image classification datasets. Our last project(Chapter 6) proposed five different weighted regularization techniques to repair group-level errors(confusion and bias errors) in DNN based software. Our results show that all these five different weighted

regularization techniques can effectively fix group-level errors with minimal cost for different scenarios. Finally, we summarized our contribution and discussed some open problems and potential future exploration.

Collectively, these projects promote the emerge and advancement of SE for AI area and contribute to the assuring of functionality, safety as well as fairness for DNN based software. We summarize the contributions of main published papers in the following:

- Presented a neuron coverage guided test generation techniques for DNN based software. Leveraged realistic image transformations to synthesize realistic corner cases for self-driving car models. Applied domain-specific metamorphic relations to generating oracle for these new generated corner cases. Implemented these techniques in DeepTest and applied it on three top performing self-driving car models in Udacity self-driving car challenge. DeepTest identified thousands of erroneous behaviours that may lead to potential fatal crash.[53]

- Conducted an empirical study on per-point robustness of DNNs under natural variation. Found that specific weak points near decision boundary will result in erroneous behaviours under natural variations. Designed and implemented a white-box approach(DEEPROBUST-W) and a black-box approach(DEEPROBUST-B) to identify these weak data points for DNN based software. Evaluated our approaches to testing 9 DNN based image classifiers and 3 DNN based self-driving car models. Our results show that DEEPROBUST-W and DEEPROBUST-B are able to achieve an F1 score of up to 91.4% and 99.1%, respectively in testing DNN based image classifiers. DEEPROBUST-W is effective in identifying weak data points with F1 score up to 78.9% in testing DNN based self-driving car models.[76]

- Introduced group-level errors, which is another type of errors DNN based software suffers, different from instance-wise errors. Categorized group-level errors into confusion errors and bias errors based on real-world reports. Proposed neuron coverage based distance metric to test DNN based software for confusion errors and bias errors. Implemented these techniques in DeepInspect and applied it to six different popular image classification datasets and eight

pretrained DNN models, including three robust trained models. It identified confusion errors with precision up to 100% (avg. 72.6%) bias errors up to 84.3% (avg. 66.8%). It found hundreds of classification mistakes in widely used models, many exposing errors indicating confusion or bias.[78]

- Proposed five weighted regularization techniques to automatically repair confusion errors and bias errors for DNN based software. These five weighted regularization techniques function at different phases of retraining or inference of DNNs to make the model take more effort in learning from target classes. These techniques can be applied in different scenarios including even retraining is not possible. Implemented these techniques in WR and applied it to four widely-used datasets and architectures. The results show that WR can effectively fix confusion errors and bias errors with minimal cost of the overall performance.

7.2 Limitation and Future Work

- Our techniques proposed in DeepTest leverage neuron coverage to measure the quality of generated test cases and how well DNN based software are tested. Future works may include the exploration and comparison of other coverage techniques such as neuron set and neuron path coverage, *etc.* for DNN based software. We proposed neuron coverage guided test generation techniques. In future, other search based fuzzing techniques can be explored and compared. In DeepTest, we specifically applies our implementation on DNN based self-driving cars models. These models only have camera input and output steering angles. Future testing of DNN based self-driving car may handle more sensors' inputs and other outputs such as brake, acceleration control, *etc.* besides steering angles.

- We studied per-point robustness of DNNs under natural variation in DeepRobust. We observed that specific weak points are more likely to fail a DNN based software under natural transformations. Then, we proposed a white-box approach(DEEPROBUST-W) and a black-box approach(DEEPROBUST-B) to identify these weak points for DNN based image classi-

fiers and self-driving cars. The results show that they can effectively identify weak/strong points with high precision and recall. Future work may include the exploration of other consistency analysis methods [238] such as variation ratio, entropy *etc.* We can also prioritize test cases based on identified weak points[230, 231] or retrain DNNs with weak points to improve DNNs' robustness [55].

- We introduce group-level errors in DeepInspect, orthogonal to instance-wise errors for DNN based software. There are very few existing works studying group-level errors. In future, we expect more exploration such as white-box based interpretation and debugging techniques for group-level errors for DNN based software. In future other types of errors could be explored and discovered, besides group-level errors and instance-wise errors. As more and more DNN based software are developed and deployed in real world, we expect that the fairness issue becomes more serious and our proposed bias errors will be investigated and studied in more depth.

- In our last project, we proposed five different weighted regularization techniques to repair group-level errors. We evaluated these techniques on popular single-label and multi-label DNN based image classification. Future works may include repairing other applications such as object recognition and recommending systems, *etc.* We proposed weighted regularization in different stages of retraining or inference of DNNs to repair group-level errors. In future other DNN repairing techniques such as weights updates and layers adding/removing, *etc.* can be studied and compared. We also expect that increasing number of works published in SE for AI area will focus on repairing DNN based software.

References

[1] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv: 1605.07146*, 2016.

[2] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[3] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*, PMLR, 2019, pp. 6105–6114.

[4] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53 040–53 065, 2019.

[5] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[6] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 342–357.

[7] H. Hemmati, "How effective are code coverage criteria?" In *2015 IEEE International Conference on Software Quality, Reliability and Security*, IEEE, 2015, pp. 151–156.

[8] I. H. Osman and J. P. Kelly, "Meta-heuristics theory and applications," *Journal of the Operational Research Society*, vol. 48, no. 6, pp. 657–657, 1997.

[9] P. McMinn, "Search-based software test data generation: A survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[10] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing*, vol. 5, no. 3, pp. 315–331, 2005.

[11] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[12] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2009.

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *ACM Sigplan Notices*, ACM, vol. 40, 2005, pp. 213–223.

[14] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*, Springer, 2006, pp. 419–423.

[15] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *International conference on tools and algorithms for the construction and analysis of systems*, Springer, 2007, pp. 134–138.

[16] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, 2008, pp. 151–166.

[17] C. Cadar, D. Dunbar, D. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[18] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011, pp. 1066–1071.

[19] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 189–200.

[20] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of metamorphic testing to supervised classifiers," in *2009 Ninth International Conference on Quality Software*, IEEE, 2009, pp. 135–144.

[21] C. Murphy, *Metamorphic testing techniques to detect defects in applications without test oracles*. Columbia University, 2010.

[22] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[23] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[24] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 947–954.

[25] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 3–13.

[26] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[27] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[29] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 313–316.

[30] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010, pp. 65–74.

[31] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Software: practice and experience*, vol. 43, no. 9, pp. 1121–1142, 2013.

[32] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 752–762.

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

[34] F. Logozzo and T. Ball, "Modular and Verified Automatic Program Repair," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, Tucson, Arizona, USA: ACM, 2012, pp. 133–146, ISBN: 978-1-4503-1561-6.

[35] F. Logozzo and M. Martel, "Automatic repair of overflowing expressions with abstract interpretation," *arXiv preprint arXiv:1309.5148*, 2013.

[36] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 459–470.

[37] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert, "Automated generation of buffer overflow quick fixes using symbolic execution and smt," in *International Conference on Computer Safety, Reliability, and Security*, Springer, 2014, pp. 441–456.

[38] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, 2017.

[39] G. Yang, K. Min, and B. Lee, "Applying deep learning algorithm to automatic bug localization and repair," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1634–1641.

[40] *This is how bad self-driving cars suck in the rain*, `http://jalopnik.com/this-is-how-bad-self-driving-cars-suck-in-the-rain-1666268433`, 2014.

[41] *Who's responsible when an autonomous car crashes?* `http://money.cnn.com/2016/07/07/technology/tesla-liability-risk/index.html`, 2016.

[42] *Tesla's self-driving system cleared in deadly crash*, `https://www.nytimes.com/2017/01/19/business/tesla-model-s-autopilot-fatal-crash.html`, 2017.

[43] *Google's self-driving car caused its first crash*, `https://www.wired.com/2016/02/googles-self-driving-car-may-caused-first-crash/`, 2016.

[44] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[45] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *International Conference of Software Engineering (ICSE), 2018 IEEE conference on*, IEEE, 2018.

[46] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic autonomous driving system testing," *arXiv preprint arXiv:1802.02295*, 2018.

[47] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, "Exploring the landscape of spatial robustness," in *International Conference on Machine Learning*, 2019, pp. 1802–1811.

[48] F. Yang, Z. Wang, and C. Heinze-Deml, "Invariance-inducing regularization using worst-case transformations suffices to boost accuracy and spatial robustness," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 14 757–14 768.

[49] M. Balunovic, M. Baader, G. Singh, T. Gehr, and M. Vechev, "Certifying geometric robustness of neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 287–15 297.

[50] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," ASE 2018, pp. 120–131, 2018.

[51] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[52] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[53] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.

[54] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[55] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020, pp. 1147–1158.

[56] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2018, pp. 100–111.

[57] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: ACM, 2018, pp. 109–119, ISBN: 978-1-4503-5937-5.

[58] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: Testing and debugging deep neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2019, pp. 111–114.

[59] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[60] Z. Zhong, Y. Tian, and B. Ray, "Understanding local robustness of deep neural networks under natural variations," *Fundamental Approaches to Software Engineering*, vol. 12649, p. 313, 2021.

[61] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[62] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020, pp. 1135–1146.

[63] X. Ren, B. Yu, H. Qi, F. Juefei-Xu, Z. Li, W. Xue, L. Ma, and J. Zhao, "Few-shot guided mix for dnn repairing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 717–721.

[64] J. Sohn, S. Kang, and S. Yoo, "Search based repair of deep neural networks," *arXiv preprint arXiv:1912.12463*, 2019.

[65] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: Automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: ACM, 2018, pp. 175–186, ISBN: 978-1-4503-5573-5.

[66] S. Wang, Y. Chen, A. Abdou, and S. Jana, "Mixtrain: Scalable training of formally robust neural networks," *arXiv preprint arXiv:1811.02625*, 2018.

[67] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I"*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 97–117.

[68] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *International Conference on Machine Learning*, 2018, pp. 3575–3583.

[69] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter, "Scaling provable adversarial defenses," in *Advances in Neural Information Processing Systems*, 2018, pp. 8400–8409.

[70] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," 2018.

[71] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, "A rotation and a translation suffice: Fooling cnns with simple transformations," in *Proceedings of the 36th international conference on machine learning (ICML)*, 2019.

[72] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations (ICLR)*, 2018.

[73] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv preprint arXiv:1705.07204*, 2017.

[74] X. Ma, B. Li, Y. Wang, S. M. Erfani, S. Wijewickrema, G. Schoenebeck, D. Song, M. E. Houle, and J. Bailey, "Characterizing adversarial subspaces using local intrinsic dimensionality," in *International Conference on Learning Representations (ICLR)*, 2018.

[75] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 478–489.

[76] R. B. Zhong Z Tian Y, "Understanding local robustness of deep neural networks under natural variations," in *Fundamental Approaches to Software Engineering*, 2021.

[77] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song, "Robust physical-world attacks on machine learning models," *arXiv preprint arXiv:1707.08945*, 2017.

[78] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray, "Testing dnn image classifier for confusion & bias errors," in *International Conference of Software Engineering (ICSE)*, 2020.

[79] L. Grush, "Google engineer apologizes after photos app tags two black people as gorillas," 2015.

[80] A. Rosenfeld, R. Zemel, and J. K. Tsotsos, "The elephant in the room," *arXiv preprint arXiv:1808.03305*, 2018.

[81] MalletsDarker, "I took a few shots at lake louise today and google offered me this panorama," 2018.

[82] A. Rose, "Are face-detection cameras racist?," 2010.

[83] J. Zhao, T. Wang, M. Yatskar, V. Ordonez, and K.-W. Chang, "Men also like shopping: Reducing gender bias amplification using corpus-level constraints," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 2941–2951.

[84] J. Buolamwini and T. Gebru, "Gender shades: Intersectional accuracy disparities in commercial gender classification," in *FAT*, 2018.

[85] Y. Tian, "Repairing confusion and bias errors for dnn-based image classifiers," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1699–1700.

[86] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at odds with accuracy," in *International Conference on Learning Representations (ICLR)*, 2019.

[87] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[88] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, no. 4, pp. 339–353, 2009.

[89] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. 2016, Book in preparation for MIT Press.

[90] A. Karpathy, *Convolutional neural networks*, `http://cs231n.github.io/convolutional-networks/`.

[91] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[92] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[93] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st. Boca Raton, FL, USA: CRC Press, Inc., 1999.

[94] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, *et al.*, *Gradient flow in recurrent nets: The difficulty of learning long-term dependencies*, 2001.

[95] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[96] *Baidu apollo*, `https://github.com/ApolloAuto/apollo`, 2017.

[97] *Tesla autopilot*, `https://www.tesla.com/autopilot`, 2016.

[98] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in neural information processing systems*, 2007, pp. 153–160.

[99] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[100] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[101] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[102] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.

[103] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, May 2012.

[104] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[105] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, Springer, 2014, pp. 740–755.

[106] M. Yatskar, L. Zettlemoyer, and A. Farhadi, "Situation recognition: Visual semantic role labeling for image understanding," in *Conference on Computer Vision and Pattern Recognition*, 2016.

[107] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[108] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," *arXiv preprint arXiv: 1803.04792*, 2018.

[109] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–20, 2019.

[110] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "No need to worry about adversarial examples in object detection in autonomous vehicles," in *Spotlight Oral Workshop at Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[111] A. Raghunathan, J. Steinhardt, and P. Liang, "Certified defenses against adversarial examples," *6th International Conference on Learning Representations (ICLR)*, 2018.

[112] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, A. Kurakin, R. Sheatsley, *et al.*, "Cleverhans v2. 0.0: An adversarial machine learning library," *arXiv preprint arXiv:1610.00768*, 2016.

[113] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[114] J. Kos, I. Fischer, and D. Song, "Adversarial examples for generative models," *arXiv preprint arXiv:1702.06832*, 2017.

[115] N. Narodytska and S. P. Kasiviswanathan, "Simple black-box adversarial perturbations for deep networks," in *Workshop on Adversarial Training, NIPS 2016*, 2016.

[116] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 427–436.

[117] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ACM, 2017, pp. 506–519.

[118] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 372–387.

[119] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations (ICLR)*, 2014.

[120] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," *arXiv preprint arXiv:1702.02284*, 2017.

[121] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Workshop track at International Conference on Learning Representations (ICLR)*, 2017.

[122] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Advances in Neural Information Processing Systems*, 2016, pp. 2613–2621.

[123] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 39–57.

[124] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Detecting adversarial samples from artifacts," *arXiv preprint arXiv:1703.00410*, 2017.

[125] K. Grosse, P. Manoharan, N. Papernot, M. Backes, and P. McDaniel, "On the (statistical) detection of adversarial examples," *arXiv preprint arXiv:1702.06280*, 2017.

[126] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[127] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," in *International Conference on Learning Representations (ICLR)*, 2017.

[128] N. Papernot and P. McDaniel, "Extending defensive distillation," *arXiv preprint arXiv: 1705.05264*, 2017.

[129] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Security and Privacy (SP), 2016 IEEE Symposium on*, IEEE, 2016, pp. 582–597.

[130] U. Shaham, Y. Yamada, and S. Negahban, "Understanding adversarial training: Increasing local stability of neural nets through robust optimization," *arXiv preprint arXiv:1511.05432*, 2015.

[131] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[132] S. Zheng, Y. Song, T. Leung, and I. Goodfellow, "Improving the robustness of deep neural networks via stability training," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4480–4488.

[133] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, "Adversarial example defenses: Ensembles of weak defenses are not strong," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, ser. WOOT'17, Vancouver, BC, Canada: USENIX Association, 2017, pp. 15–15.

[134] B. T. Luong, S. Ruggieri, and F. Turini, "K-nn as an implementation of situation testing for discrimination discovery and prevention," in *Proceedings of the 17th ACM SIGKDD*

international conference on Knowledge discovery and data mining, ACM, 2011, pp. 502–510.

[135] R. Zemel, Y. Wu, K. Swersky, T. Pitassi, and C. Dwork, "Learning fair representations," in *Proceedings of the 30th International Conference on Machine Learning*, 2013, pp. 325–333.

[136] M. B. Zafar, I. Valera, M. Gomez Rodriguez, and K. P. Gummadi, "Fairness constraints: Mechanisms for fair classification," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, ser. (AISTATS) 2017, vol. 54, JMLR, 2017.

[137] Y. Brun and A. Meliou, "Software fairness," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: ACM, 2018, pp. 754–759, ISBN: 978-1-4503-5573-5.

[138] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. S. Zemel, "Fairness through awareness," *In Proceedings of the Innovations in Theoretical Computer Science Conference*, vol. abs/1104.3913, pp. 214–226, 2012.

[139] M. Hardt, E. Price, and N. Srebro, "Equality of opportunity in supervised learning," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, Barcelona, Spain, 2016, pp. 3323–3331, ISBN: 978-1-5108-3881-9.

[140] S. Barocas, M. Hardt, and A. Narayanan, *Fairness and Machine Learning*. fairmlbook.org, 2018, `http://www.fairmlbook.org`.

[141] A. K. Menon and R. C. Williamson, "The cost of fairness in binary classification," in *Conference on Fairness, Accountability and Transparency, FAT 2018, 23-24 February 2018, New York, NY, USA*, 2018, pp. 107–118.

[142] M. Donini, L. Oneto, S. Ben-David, J. Shawe-Taylor, and M. Pontil, "Empirical risk minimization under fairness constraints," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, 2018, pp. 2796–2806.

[143] A. L. Lamy, Z. Zhong, A. K. Menon, and N. Verma, "Noise-tolerant fair classification," *CoRR*, vol. abs/1901.10837, 2019. arXiv: `1901.10837`.

[144] M. J. Kusner, J. Loftus, C. Russell, and R. Silva, "Counterfactual fairness," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 4066–4076.

[145] M. P. Kim, O. Reingold, and G. N. Rothblum, "Fairness through computationally-bounded awareness," *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, 2018.

[146] T. Calders, F. Kamiran, and M. Pechenizkiy, "Building classifiers with independency constraints," in *2009 IEEE International Conference on Data Mining Workshops*, IEEE, 2009, pp. 13–18.

[147] M. B. Zafar, I. Valera, M. Gomez Rodriguez, and K. P. Gummadi, "Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment," in *Proceedings of the 26th International Conference on World Wide Web*, Perth, Australia, 2017, pp. 1171–1180.

[148] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: Testing software for discrimination," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 498–510.

[149] *The numbers don't lie: Self-driving cars are getting good*, https://www.wired.com/2017/02/california-dmv-autonomous-car-disengagement, 2017.

[150] C. DMV, *Autonomous vehicle disengagement reports*, https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2016, 2016.

[151] *Autonomous vehicles enacted legislation*, http://www.ncsl.org/research/transportation/autonomous-vehicles-self-driving-vehicles-enacted-legislation.aspx, 2017.

[152] *Google auto waymo disengagement report for autonomous driving*, https://www.dmv.ca.gov/portal/wcm/connect/946b3502-c959-4e3b-b119-91319c27788f/GoogleAutoWaymo_disengage_report_2016.pdf?MOD=AJPERES, 2016.

[153] *Inside waymo's secret world for training self-driving cars*, https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/, 2017.

[154] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," 2014.

[155] *Software 2.0*, https://medium.com/@karpathy/software-2-0-a64152b37c35, 2017.

[156] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *International Conference on Computer Aided Verification*, Springer, 2017, pp. 3–29.

[157] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *Computer Aided Verification*, Springer, 2010, pp. 243–257.

[158] *Udacity self driving car challenge 2*, https://github.com/udacity/self-driving-car/tree/master/challenges/challenge-2, 2016.

[159] S. Hijazi, R. Kumar, and C. Rowen, "Using convolutional neural networks for image recognition," Tech. Rep., 2015.[Online]. Available: http://ip. cadence. com/uploads/901/cnn-wp-pdf, Tech. Rep., 2015.

[160] *Affine transformation*, https://www.mathworks.com/discovery/affine-transformation.html, 2015.

[161] *Affine transformations*, http://docs.opencv.org/3.1.0/d4/d61/tutorial_warp_affine.html, 2015.

[162] *The opencv reference manual*, 2.4.9.0, Apr. 2014.

[163] *Add dramatic rain to a photo in photoshop*, https://design.tutsplus.com/tutorials/add-dramatic-rain-to-a-photo-in-photoshop--psd-29536, 2013.

[164] *How to create mist: Photoshop effects for atmospheric landscapes*, http://www.techradar.com/how-to/photography-video-capture/cameras/how-to-create-mist-photoshop-effects-for-atmospheric-landscapes-1320997, 2013.

[165] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep., 1998.

[166] *Rambo model*, https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/rambo, 2016.

[167] *Chauffeur model*, https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur, 2016.

[168] *Epoch model*, https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/cg23, 2016.

[169] F. Chollet *et al.*, *Keras*, https://github.com/fchollet/keras, 2015.

[170] *Udacity self driving car challenge 2 dataset*, https://github.com/udacity/self-driving-car/tree/master/datasets/CH2, 2016.

[171] *Comma.ai's steering model*, https://github.com/commaai/research/blob/master/train_steering_model.py, 2016.

[172] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[173] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[174] *Open source computer vision library*, `https://github.com/itseez/opencv`, 2015.

[175] C. Spearman, "The proof and measurement of association between two things," *The American journal of psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[176] J. Hauke and T. Kossowski, "Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data," *Quaestiones geographicae*, vol. 30, no. 2, p. 87, 2011.

[177] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, IEEE, 2016, pp. 63–74.

[178] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[179] I. Goodfellow and N. Papernot, *The challenge of verification and testing of machine learning*, `http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html`, 2017.

[180] Y. Liu, X. Chen, C. Liu, and D. X. Song, "Delving into transferable adversarial examples and black-box attacks," in *International Conference on Learning Representations (ICLR)*, 2017.

[181] M. J. Wilber, V. Shmatikov, and S. Belongie, "Can we still avoid automatic face detection?" In *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*, IEEE, 2016, pp. 1–9.

[182] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 1528–1540.

[183] P. Laskov *et al.*, "Practical evasion of a learning-based classifier: A case study," in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014, pp. 197–211.

[184] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[185] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," in *Proceedings of the 2017 European Symposium on Research in Computer Security*, 2017.

[186] H. Anderson, *Evading next-gen av using a.i.* `https://www.defcon.org/html/defcon-25/dc-25-index.html`, 2017.

[187] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting adversarial input sequences for recurrent neural networks," in *Military Communications Conference, MILCOM 2016-2016 IEEE*, IEEE, 2016, pp. 49–54.

[188] T. Miyato, A. M. Dai, and I. Goodfellow, "Adversarial training methods for semi-supervised text classification," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[189] M. Cisse, P. Bojanowski, E. Grave, Y. Dauphin, and N. Usunier, "Parseval networks: Improving robustness to adversarial examples," in *International Conference on Machine Learning*, 2017, pp. 854–863.

[190] G. Wang, T. Wang, H. Zheng, and B. Y. Zhao, "Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers.," in *USENIX Security Symposium*, 2014, pp. 239–254.

[191] J. Steinhardt, P. W. Koh, and P. Liang, "Certified defenses for data poisoning attacks," *arXiv preprint arXiv:1706.03691*, 2017.

[192] Z. Q. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004, pp. 346–351.

[193] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing.," in *SEKE*, vol. 8, 2008, pp. 867–872.

[194] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[195] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering*

in Practice, ser. ICSE-SEIP '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 291–300.

[196] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering*, IEEE Press, 2019, pp. 1039–1049.

[197] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, 1027–1038.

[198] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 477–487.

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

[200] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lü, "Boosting operational dnn testing efficiency through conditioning," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 499–509.

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

[202] E. H. Simpson, "Measurement of diversity," *nature*, vol. 163, no. 4148, pp. 688–688, 1949.

[203] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song, "Spatially transformed adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2018.

[204] U. Saxena, *Automold*, `https://github.com/UjjwalSaxena/Automold--Road-Augmentation-Library/`.

[205] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *FSE*, 2005.

[206] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms*, 2017. arXiv: `cs.LG/1708.07747 [cs.LG]`.

[207] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, May 2012.

[208] T. W. Yuval Netzer, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[209] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[210] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, "Residual attention network for image classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 3156–3164.

[211] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[212] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20, ACM, 2020, 12 pages.

[213] Udacity, *A self-driving car simulator built with Unity*, `https://github.com/udacity/self-driving-car-sim`, Online; accessed 18 August 2019, 2017.

[214] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars.," *CoRR*, vol. abs/1604.07316, 2016.

[215] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[216] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[217] S. S. Sawilowsky, "New effect size rules of thumb," *Journal of Modern Applied Statistical Methods*, vol. 8, no. 2, p. 26, 2009.

[218] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[219] C. Guo, J. Gardner, Y. You, A. G. Wilson, and K. Weinberger, "Simple black-box adversarial attacks," in *International Conference on Machine Learning*, PMLR, 2019, pp. 2484–2493.

[220] S. Moon, G. An, and H. O. Song, "Parsimonious black-box adversarial attacks via efficient combinatorial optimization," in *International Conference on Machine Learning*, PMLR, 2019, pp. 4636–4645.

[221] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *Proceedings of the 41st International Conference on Software Engineering*, IEEE Press, 2019, pp. 1245–1256.

[222] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," *arXiv preprint arXiv:1809.08098*, 2018.

[223] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," in *Advances in Neural Information Processing Systems*, 2019, pp. 478–489.

[224] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, *Adversarial examples are not bugs, they are features*, 2019.

[225] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, L. Zhang, B. Yu, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *International Conference of Software Engineering (ICSE)*, 2020.

[226] S. Gerasimou, H. F. Eniser, A. Sen, and A. Çakan, "Importance-driven deep learning system testing," in *International Conference of Software Engineering (ICSE)*, 2020.

[227] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *International Conference of Software Engineering (ICSE)*, 2020.

[228] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *International Conference of Software Engineering (ICSE)*, 2020.

[229] H. F. Eniser, S. Gerasimou, and A. Sen, "Deepfault: Fault localization for deep neural networks," in *Fundamental Approaches to Software Engineering*, R. Hähnle and W. van der Aalst, Eds., Cham: Springer International Publishing, 2019, pp. 171–191, ISBN: 978-3-030-16722-6.

[230] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, IEEE, 2019, pp. 63–70.

[231] S. Udeshi, X. Jiang, and S. Chattopadhyay, "Callisto: Entropy-based test generation and data quality assessment for machine learning systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, pp. 448–453.

[232] X. Qiu, E. Meyerson, and R. Miikkulainen, "Quantifying point-prediction uncertainty in neural networks via residual estimation with an i/o kernel," in *International Conference on Learning Representations*, 2020.

[233] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, PMLR, 2016, pp. 1050–1059.

[234] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *International Conference on Machine Learning*, PMLR, 2017, pp. 1321–1330.

[235] M. Teye, H. Azizpour, and K. Smith, "Bayesian uncertainty estimation for batch normalized deep networks," in *International Conference on Machine Learning*, PMLR, 2018, pp. 4907–4916.

[236] S. Jha, S. Raj, S. Fernandes, S. K. Jha, S. Jha, B. Jalaian, G. Verma, and A. Swami, "Attribution-based confidence metric for deep neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 11 826–11 837.

[237] H. Jiang, B. Kim, and M. Gupta, "To trust or not to trust a classifier," in *Advances in Neural Information Processing Systems*, 2018, 5541—5552.

[238] Y. Gal, "Uncertainty in deep learning," 2016.

[239] K. Pei, Y. Cao, J. Yang, and S. Jana, "Towards practical verification of machine learning: The case of computer vision systems," *arXiv preprint arXiv:1712.01785*, 2017.

[240] S. A. Seshia, A. Desai, T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue, "Formal specification for deep neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, Springer, 2018, pp. 20–34.

[241] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, Springer, 2017, pp. 269–286.

[242] P. Kamavisdar, S. Saluja, and S. Agrawal, "A survey on image classification approaches and techniques," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, no. 1, pp. 1005–1009, 2013.

[243] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*, PMLR, 2019, pp. 4901–4911.

[244] E. Wong, F. Schmidt, J. H. Metzen, and J. Z. Kolter, "Scaling provable adversarial defenses," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 8410–8419.

[245] *Base pretrained models and datasets in pytorch*, 2017.

[246] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 432–441.

[247] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models.," *JSS*, vol. 83, no. 1, pp. 2–17, 2010.

[248] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, ACM, 2013, pp. 147–157.

[249] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–10.

[250] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: Hit or miss?" In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 322–331.

[251] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 428–439.

[252] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[253] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/01621459.1952.10483441`.

[254] C. Xie, Y. Wu, L. v. d. Maaten, A. L. Yuille, and K. He, "Feature denoising for improving adversarial robustness," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 501–509.

[255] Z. C. Lipton, "The mythos of model interpretability," *Proceedings of the 33rd International Conference on Machine Learning Workshop*, 2016.

[256] Q.-s. Zhang and S.-C. Zhu, "Visual interpretability for deep learning: A survey," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 1, pp. 27–39, 2018.

[257] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[258] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing*, 2017.

[259] D. Bau, B. Zhou, A. Khosla, A. Oliva, and A. Torralba, "Network dissection: Quantifying interpretability of deep visual representations," in *Computer Vision and Pattern Recognition*, 2017.

[260] Y. Dong, H. Su, J. Zhu, and F. Bao, "Towards interpretable deep neural networks by leveraging adversarial examples," *arXiv preprint arXiv:1708.05493*, 2017.

[261] N. Papernot and P. McDaniel, "Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning," *arXiv preprint arXiv:1803.04765*, 2018.

[262] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, 2. MIT press Cambridge, 2016, vol. 1.

[263] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1135–1146.

[264] M. Hardt, E. Price, and N. Srebro, "Equality of opportunity in supervised learning," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16, Red Hook, NY, USA: Curran Associates Inc., 2016, 3323–3331, ISBN: 9781510838819.

[265] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *International Conference on Computer Vision (ICCV)*, 2019, published.

[266] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 614–618.