Assessing the Reliability of Deep Learning Applications

by

Yongqiang Tian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2023

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Zhi Jin Professor, Dept. of Computer Science and Technology Peking University
Supervisor:	Chengnian Sun Assistant Professor, Cheriton School of Computer Science University of Waterloo
Co-Supervisor:	Shing-Chi Cheung Chair Professor, Dept. of Computer Science and Engineering The Hong Kong University of Science and Technology
Internal Member:	Meng Xu Assistant Professor, Cheriton School of Computer Science University of Waterloo
Internal Member:	Raymond Wong Professor, Dept. of Computer Science and Engineering The Hong Kong University of Science and Technology
Internal-External Member:	Ross Murch Chair Professor, Dept. of Electronic and Computer Engineering The Hong Kong University of Science and Technology

Author's Declaration

Statement of Contributions

This thesis contains the following three published or accepted studies. As the first author, I led each study and made major contributions to each study, including but not limited to (1) proposing ideas, (2) conducting literature review and preliminary studies, (3) developing and implementing techniques, (4) conducting experiments and analysis, (5) drafting manuscripts and responding to review comments, and (6) presenting the work in conference if any. The co-authors of each study provided feedback to improve the quality of each study.

- **To what extent do DNN-based image classification models make unreliable inferences?**
  Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang.
  Empirical Software Engineering (2021).

This study contributes to the **Chapter 3** of this thesis. Prof. Shiqing Ma and Prof. Ming Wen provided feedback to improve the methodology. Prof. Ming Wen, Prof. Yepang Liu, and Prof. Xiangyu Zhang helped me polish the writing and organization. Prof. Shing-Chi Cheung supervised the entire study and provided suggestions on each part.

- **Finding Deviated Behaviors of the Compressed DNN Models for Image Classifications**
  Yongqiang Tian, Wuqi Zhang, Ming Wen, Shing-Chi Cheung, Chengnian Sun, Shiqing Ma, and Yu Jiang.
  ACM Transactions on Software Engineering and Methodology, accepted in 2023.

This study contributes to the **Chapter 4** of this thesis. Mr. Wuqi Zhang helped me collect part of the compressed models and conduct some experiments using baselines. Prof. Ming Wen helped me polish the writing of this study. Prof. Shiqing Ma and Prof. Yu Jiang provided feedback on this study. Prof. Chengnian Sun and Prof. Shing-Chi Cheung supervised the entire study and provided suggestions on each part.

- **Revisiting the Evaluation of Deep Learning-Based Compiler Testing**
  Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung.
  The 32nd International Joint Conference on Artificial Intelligence (IJCAI), accepted in 2023.

This study contributes to the **Chapter 5** of this thesis. Mr. Zhenyang Xu helped me in the implementation of Kitten. Mr. Zhenyang Xu, Mr. Yiwen Dong and Prof. Shing-Chi

Cheung provided feedback on the writing of this study. Prof. Chengnian Sun supervised the entire study and provided suggestions on each part.

Abstract

Deep Learning (DL) applications are widely deployed in diverse areas, such as image classification, natural language processing, and auto-driving systems. Although these applications achieve outstanding performance in certain metrics like accuracy, developers have raised strong concerns about their reliability since the logic of DL applications is a black box for humans. Specifically, DL applications learn their logic during stochastic training and encode it in high-dimensional weights of DL models. Unlike source code in conventional software, such weights are infeasible for humans to directly interpret, examine, and validate. As a result, the reliability issues in DL applications are not easy to detect and may cause catastrophic accidents in safety-critical missions. Therefore, it is critical to adequately assess the reliability of DL applications.

This thesis aims to help software developers assess the reliability of DL applications from the following three perspectives.

The first study proposes *object-relevancy*, a property that reliable DL-based image classifiers should comply with, *i.e.*, the classification results should be made based on the features relevant to the target object in a given image, instead of irrelevant features such as the background. This study further proposes an automatic approach based on two metamorphic relations to assess if this property is violated in the image classifications. The evaluation shows that the proposed approach can effectively detect unreliable inferences violating the object-relevancy property, with an average precision of 64.1% and 96.4% for the two relations, respectively. The subsequent empirical study reveals that such unreliable inferences are prevalent in the real world and the existing training strategies cannot tackle this issue effectively.

The second study concentrates on the reliability issues induced by DL model compression. DL model compression can significantly reduce the sizes of Deep Neural Network (DNN) models, and thus facilitate the deployment of sophisticated, sizable DNN models. However, the prediction results of compressed models may deviate from those of their original models, resulting in unreliably deployed DL applications. To help developers thoroughly assess the impact of model compression, it is essential to test these models to find any *deviated behaviors* before dissemination. This study proposes DFLARE, a novel, search-based, black-box testing technique. The evaluation shows that DFLARE constantly outperforms the baseline in both efficacy and efficiency. More importantly, the triggering inputs found by DFLARE can be used to repair up to 48.48% of deviated behaviors.

The third study reveals the unreliable assessment of DL-based Program Generators (DLGs) in compiler testing. To effectively test compilers, DLGs are proposed to automatically generate massive testing programs. However, after thorough analysis of the characteristics of DLGs, this study found that the assessment of these DLGs is unfair and unreliable, since the chosen baselines, *i.e.*, Language-Specific Program Generators (LSGs), are different from DLGs in many aspects. Furthermore, this study proposed Kitten, a simple, fair, and non-DL-based baseline for DLGs. The experiments show that DLGs cannot even compete against such a simple baseline and the claimed advantages of DLGs are likely due to the biased selection of the baseline. Specifically, Kitten triggers 1,750 hang bugs and 34 distinct crashes in 72-hours of testing on GCC, while the the-state-of-art DLG only triggers 3 hang bugs and 1 distinct crash. Moreover, the code coverage achieved by Kitten is at least 2x as of that achieved by the the-state-of-art DLG.

Acknowledgements

I would like to thank all those who advised me, helped me, supported me, and collaborated with me during my Ph.D. study. The actual Ph.D. life I have experienced in the last six years contains much more pains and sufferings than what I imagined in March 2017. Fortunately, with their generous support and encouragement, I have been able to finish this special journey.

First and foremost, I am deeply grateful for the invaluable support and guidance provided by my supervisors, Prof. Shing-Chi Cheung and Prof. Chengnian Sun. They taught me the fundamental skills to conduct research, especially the critical thinking and academic writing. Their philosophy in research always motivates me to pursue solid and practical techniques for software developers. I also appreciate the opportunities offered by them to explore other research topics. Besides, I sincerely thank their support and advice for my personal life.

I sincerely thank my thesis committee members, Prof. Zhi Jin, Prof. Ross Murch, Prof. Raymond Wong, and Prof. Meng Xu, for reviewing my thesis and attending my defense. Their valuable comments and suggestions helped me polish this thesis.

My special gratitude goes to Prof. Yu Jiang and his group at Tsinghua University. During the COVID-19 pandemic, they kindly offered me a warm visit, which is an extraordinary experience in my life. I learned a lot of knowledge from them.

I am also grateful to my collaborators. Prof. Shiqing Ma, Prof. Ming Wen, and Prof. Yepang Liu gave me many practical suggestions in my research, especially the writing. I would also thank other co-authors in my publications. The collaboration with them expanded my research interests. I would like to thank all anonymous reviewers since their constructive comments and suggestions improved the quality of my work.

I gratefully acknowledge the help and support from my labmates at HKUST and UWaterloo. My special thanks go to Mr. Wuqi Zhang, Mr. Jiarong Wu, Mr. Zhenyang Xu, and Prof. Lili Wei. They helped me to go through the hardest time in my Ph.D. life. Much gratitude is owed to Mr. Yiwen Dong, Dr. Wenhan Zhu, and Mr. Xueyan Zhang for their help at Waterloo. I sincerely wish they have a good achievement in their career.

I also would like to thank the technical staff at CS System of HKUST CSE. They provided me with stable and reliable computation resources for my research. My gratitude also goes to Miss Ada Wong and other staff at HKUST SENG, as well as the staff at the CS Graduate Office, MGO and GSPA of UWaterloo. They helped me a lot to prepare the paperwork for my special dual Ph.D. program.

I sincerely thank all my family for their unconditional support and love. My parents Mr. Weiguo Tian and Miss Lifang Cheng, my sister Miss Yongjiao Tian, and my grandparents always support me to pursue my dream from my primary school. Without them, I would not be able to finish this degree. My special thanks go to my aunt Dr. Yuan Zhang and uncle Mr. Hang Chu. They gave me a lot of help and advice in my career and life. Moreover, I am profoundly thankful for the unwavering encouragement and support from my girlfriend, Miss Yao Feng. Her encouragement gives me the faith to complete this journey.

The studies in this thesis are supported by multiple research grants. At University of Waterloo, the studies in this thesis are supported by Cisco Research Gift, Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, CFI-JELF Project #40736, and GO-Bell scholarship. At HKUST, I am primarily supported by Hong Kong Ph.D. Fellowship Scheme. The studies in my thesis are also supported by Hong Kong RGC/RIF (Grant No. R5034-18), Hong Kong ITF (Grant No: MHP/055/19), Hong Kong RGC/GRF (Grant No: 16207120), MSRA Collaborative Research Grant, Microsoft Cloud Research Software Fellow Award 2019, and National Natural Science Foundation of China (Grant No. 61932021).

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DLG Deep Learning-based Program Generator

LSG Language-Specific Program Generator

API Application Programming Interface

DL Deep Learning

DNN Deep Neural Network

IoT Intenet of Things

MR Metamorphic Relation

RNN Recurrent Neural Network

Chapter 1

Introduction

This chapter introduces the motivation of this thesis, the thesis statement, and the three studies contributing to the thesis statement. It also highlights the contributions of this thesis and presents the organization.

1.1 Motivation

Over the past decade, Deep Learning (DL) applications have attracted tremendous attention from both academia and industry. By leveraging the sophisticated Deep Neural Network (DNN) models trained by massive data, these applications are achieving outstanding performances in many tasks [160, 65, 142, 64, 41, 101]. For example, in image classification, DL applications have outperformed humans in classification accuracy [65, 151]. In the Go game, AlphaGo defeated the topmost human player in the world in 2017.

Despite the success of DL applications in terms of accuracy, the **reliability** of DL applications have become one of the strongest concerns for developers [56, 144, 149, 107]. For example, a DL-based auto-driving system may fail to predict the correct steering direction when the weather conditions have changed from snowy to rainy [130, 172]. Such unreliable DL applications can cause catastrophic consequences and threaten human life, especially in mission-critical scenarios, such as auto-driving systems and malware detection [83]. Therefore, it is important to help software developers assess the reliability of DL applications.

However, it is challenging to assess the reliability of DL applications. First, DL applications are driven by data instead of source code, and such a special computing paradigm induces many previously unknown reliability issues, such as trustworthiness [56, 167] and fairness [2, 209]. Without domain-specific knowledge and comprehensive analysis, it is hard to reveal such issues. Second, even if software developers are aware of certain reliability issues, it is hard for them to assess the extent to which DL applications are affected by such issues. Unlike conventional software where the logic is written in source code, DL applications encode the logic in the weights of the DNN models during stochastic training. Such weights are complicated in terms of both dimension and amount, resulting in insurmountable difficulties for developers to interpret and examine the encoded logic. As a result, many existing approaches for software quality assurance, such as code review [10] and program analysis [122], are not applicable for DL applications [207], since these activities require developers to understand the logic of software. Therefore, to help developers assess the reliability of DL applications, the proposal of new methodologies is highly desirable.

1.2 Overview

This thesis includes three studies to help developers assess the reliability of DL applications. To be precise, the thesis statement is:

Thesis Statement. This thesis aims to help software developers assess the reliability of DL applications, by proposing effective and efficient techniques, conducting empirical studies, and providing actionable advice.

These three studies contribute to this statement from three perspectives, *i.e.*, the unreliable inference within DL applications, the unreliable deployment of DL applications, and the unreliable assessment of DL applications, respectively. This thesis focuses on these three perspectives since they correspond to the three important stages of software development life cycle [38],*i.e.*, development, deployment, and assessment.

Unreliable Inference within DL Applications. The first study proposes *object-relevancy*, a property that reliable DL-based image classifiers should comply with, *i.e.*, the classification results should be made based on the features relevant to the target object in a given image, instead of irrelevant features such as the background. This study proposes a metamorphic testing approach to assess if this property is violated in image classifications. Specifically, this study proposes two metamorphic relations (MRs) to detect such unreliable inferences. These relations expect (a) the classification results with different labels or the same labels but less certainty from models after corrupting the relevant features of images, and (b) the classification results with the same labels after corrupting irrelevant features. The inferences that violate the metamorphic relations are regarded as unreliable. The evaluation shows that the proposed approach can effectively detect unreliable inferences violating the object-relevancy property, with an average precision of 64.1% and 96.4% for the two MRs in single-label image classification models. The empirical study using our approach reveals that such unreliable inferences are prevalent in the real world and the existing training strategies cannot tame this issue effectively. This study [171] was published in *Empirical Software Engineering* in 2021 and presented in *the 44th International Conference on Software Engineering* (ICSE 2022).

- **To what extent do DNN-based image classification models make unreliable inferences?**
  Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang.
  Empirical Software Engineering, published in 2021.

Unreliable Deployment of DL Applications. The second study concentrates on the reliability issues due to model compression, a common stage for model deployment. Model compression can significantly reduce the sizes of DNN models, and thus facilitate the dissemination of sophisticated, sizable DNN models. However, the prediction results of compressed models may deviate from those of their original models, resulting in unreliable DL applications in deployment. To help developers thoroughly understand the impact of model compression, it is essential to test these models to find any *deviated behaviors* before dissemination. However, this is a non-trivial task because the architectures and gradients of compressed models are usually not available. This study proposes DFLARE, a novel, search-based, black-box testing technique. DFLARE iteratively applies a series of mutation operations to a given seed input, until a triggering input is found. For better efficacy and efficiency, DFLARE models the search problem as Markov Chains and leverages the Metropolis-Hasting algorithm to guide the selection of mutation operators in each iteration. Further, DFLARE utilizes a novel fitness function to prioritize the mutated inputs that either cause large differences between the outputs of two models, or trigger previously unobserved models' probability vectors. The evaluation results show that DFLARE constantly outperforms the baseline in terms of efficacy and efficiency: DFLARE is 17.84x∼446.06x as fast as the baseline in terms of time; the number of queries required by DFLARE to find one triggering input is only 0.186%∼1.937% of those issued by the baseline. This study also demonstrates that the triggering inputs found by DFLARE can be used to repair up to 48.48% deviated behaviors. The study [175] was accepted by *ACM Transactions on Software Engineering and Methodology* (TOSEM) in 2023.

- **Finding Deviated Behaviors of the Compressed DNN Models for Image Classifications**
  Yongqiang Tian, Wuqi Zhang, Ming Wen, Shing-Chi Cheung, Chengnian Sun, Shiqing Ma, and Yu Jiang.
  ACM Transactions on Software Engineering and Methodology, published in 2023.

Unreliable Assessment of DL Applications. The third study focuses on the unreliable assessment of DL-based compiler testing techniques. To comprehensively test compilers such as GCC and Clang, Deep Learning-based program generators (DLGs) have recently been proposed to automatically generate new programs via learning the semantics of programs. Some recent work claims that DLGs outperform Language-Specific Program Generators (LSGs) crafted by domain experts in various metrics. However, by identifying the characteristics of DLGs and LSGs, this study argued that the assessment of DLGs using LSGs is unfair, which may result in biased conclusions. To help developers reliably evaluate the DLGs, this study proposes Kitten, a simple baseline that shares many common characteristics with DLGs, except that Kitten directly derives new programs using mutations instead of any DNN models. After 1,500-CPU/GPU-hour experiment and analysis, this study found that the performance of existing DLGs cannot outperform this simple baseline Kitten. In 72 hours of testing on GCC, the-state-of-art DLGs, DeepSmith, triggers 3 hang bugs and 1 distinct crash, while Kitten triggers 1,750 hang bugs and 34 distinct crashes. Moreover, the code coverage achieved by Kitten is at least 2x as much as the one generated by DeepSmith. This study [173] has been accepted by the 32nd International Joint Conference on Artificial Intelligence (IJCAI'23).

- **Revisiting the Evaluation of Deep Learning-Based Compiler Testing**
  Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung.
  The 32nd International Joint Conference on Artificial Intelligence (IJCAI'23), accepted in 2023.

1.3 Contribution

This thesis made the following contributions.

Assessing the Reliability This thesis proposed three techniques to help developers to assess the reliability of DL applications. Each technique is specially designed based on domain-specific knowledge. The first technique leverages metamorphic testing to find the inference violating the object-relevancy property. The second technique guides the search process of triggering inputs using 1) the difference between the original model and the compressed model, and 2) the observed model states. The last technique constructs a simple yet effective baseline using mutation operators to help developers conduct reliable assessments of DLGs.

Understanding the Reliability Using the proposed techniques, this thesis conducted extensive experiments with diverse DL applications. The results show that DL applications pervasively suffer from these reliability issues. Moreover, the third work of this thesis is the first study to reveal the unreliable assessment of DLGs in compiler testing.

Improving the Reliability This thesis provided actionable suggestions and techniques to help developers and researchers improve the reliability of DL applications. The first and third studies proposed many suggestions based on extensive empirical analysis. The second study proposed DREPAIR to fix the reliability issues using the triggering inputs found by DFLARE and .

1.4 Organization

The remainder of this thesis introduces the details of these three studies. Specifically, §2 introduces the background, including DL applications and the related reliability. §3 introduces the object-relevance property and our proposed metamorphic testing approach. §4 details the deviated behaviors induced by model compression and DFLARE, our effective and efficient approach to finding the trigger inputs of original models and compressed models. The unreliable assessment of DLGs is discussed in §5, together with the simple and fair baseline proposed by us. Lastly, §6 concludes this thesis and discusses future work.

Chapter 2

Preliminary

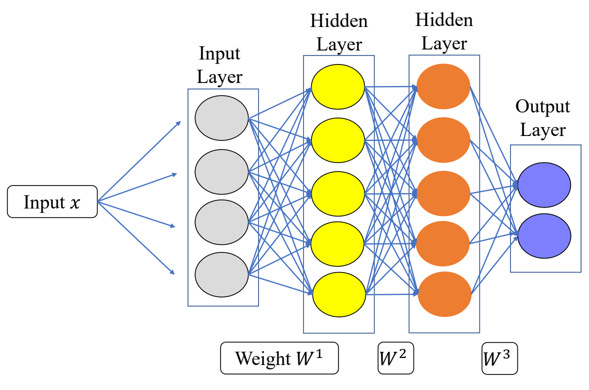
This chapter introduces the background related to this thesis, including the DL applications and their reliability.

2.1 Deep Learning Applications

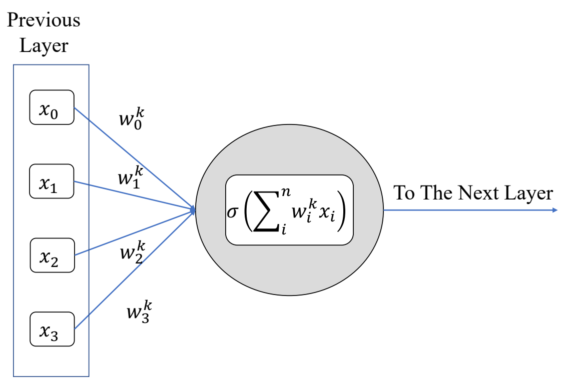
Deep Learning (DL) is one of the computing paradigms of machine leaning [87]. It leverages diverse models consisting of multiple hidden layers to capture the characteristics of data. Such DL models can be used in various scenarios, such as classification [65, 69] and detection [143]. Many model architectures are proposed to accommodate the requirements of different scenarios, including Deep Neural Network (DNN), Recurrent Neural Network (RNN) and so on.

Deep Neural Network is one of the most common model architectures in DL. As shown in Figure 2.1a, DNN models [108] usually have multiple connected hidden layers and each layer contains a collection of computing units, *i.e.* neurons. The neurons in consecutive layers are connected by weighted edges. Figure 2.1b illustrates the computation process of each neuron. Each neuron takes as input the output values from the neurons in preceding layers $(x_0, x_1, \dots)$ and corresponding weights $(w_0^k, w_1^k, \dots)$, and then produces the aggregated value $(\sum_i^k w_i^k x_i)$ to the neurons in subsequent layers. Before passing the data to the next neuron, each neuron usually processes the aggregated value using activation functions σ, such as ReLU [120] and Sigmod [112].

DL applications refer to software applications that leverage DL models to accomplish certain tasks [172, 130], such as image classification [39, 65, 160, 65], object detection [143,



(a) A DNN model.



(b) A neuron in k-th layer of DNN models

Figure 2.1: (a): A DNN model that presents the function $f = \sigma\left(W^3\left(\sigma\left(W^2\left(\sigma\left(W^1 x\right)\right)\right)\right)\right)$. W is the weight between layers and x is the input variable. σ is the activation function. (b): A neuron in DNN models and its computation process.

1], natural language processing [60, 41, 140], vulnerability detection [30, 219, 92], and so on [203]. Besides the DNN model, DL applications may also contain other conventional software components, *e.g.*, the auxiliary function for data pre-processing.

Figure 2.2 shows the workflow of a typical DL application. Developers write source code using deep learning frameworks (*e.g.*, TensorFlow [168] and PyTorch [137]) to build the structure of DL model, including the number of layers, the type of each layer and activation function, and so on. Developers also need to specify the data processing process, the training loss function, and other hyper-parameters. Then the DNN model is trained using the training dataset. During the training, the weights in DL models are updated using backward propagation [150], until its loss value and accuracy are close to saturation. When the DL applications are deployed, input will be fed into the trained DL model for inference.

2.2 The Reliability of Deep Learning Applications

Many studies discussed the reliability issues of DL applications from different perspectives [172, 130, 196, 146, 208, 107, 56]. This section briefly outlines some of them.

Quality The quality of the outputs from DL applications should be reasonable. The quality is usually measured in customized metrics for different tasks. For example,

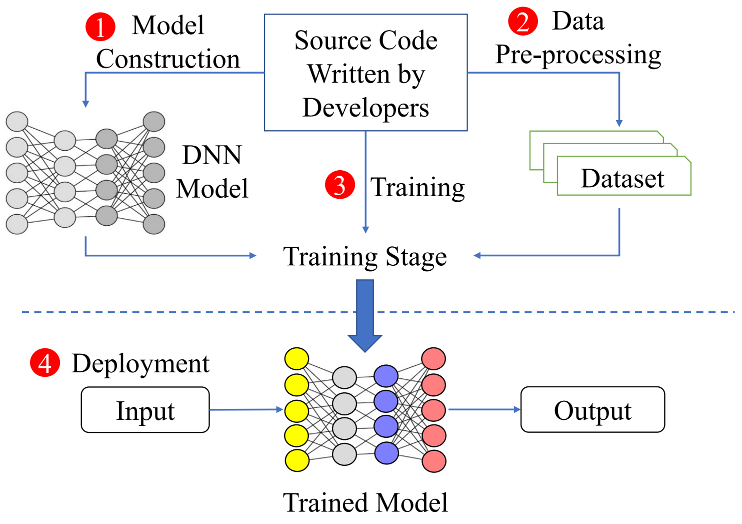


Figure 2.2: The workflow of deep learning applications.

accuracy and *precision* are commonly used in classification tasks where the inference results of classifiers should be the same as ground truths [172, 130]. Bilingual Evaluation Understudy, *a.k.a.* BLEU, is a metric widely used to evaluate the quality of machine translation [127].

Robustness The inference results should be correct, no matter how inputs are mutated or where the applications are deployed [172, 196]. For example, if the image of an apple is rotated by 90 degrees, the inference should still be "apple". If the classifier is deployed in mobile phones or Internet of Things (IoT) devices, the output should still be "apple".

Adversarial Robustness The inference results should remain unchanged when human-imperceptible perturbation is applied to inputs [22, 53]. The perturbed inputs that lead to different results from the original ones are usually referred to as *adversarial samples*. DL applications are prone to be vulnerable to adversarial attacks and there are many studies in this direction [22, 115, 53, 58, 206, 107, 4].

Trustworthiness The inference process of DL application should be trustworthy [144, 167]. For example, the logic behind the inference processes should make sense to humans. For example, given an image of an apple, a reliable DL-based image classifier should classify it as "apple" because of the appearance of an apple in this image, instead of anything irrelevant to the apple, such as the background. Backdoor attack is an effective attacking method exploiting the untrustworthiness of DL Applications [56, 210].

There are various root causes for the reliability issues including but not limited to problematic datasets [105, 163, 119], incorrect implementation [45, 195, 20, 213], defects in DL infrastructures [91, 131, 156], improper development and deployment [59, 68], and so on.

The reliability issues focused on by this thesis are in line with the aforementioned issues, and extend them. The first study concentrated on the object-relevancy property of deep learning-based image classifiers. Specifically, the inference results from these classifiers can be dominated by the background of images, instead of the objects. Such inference processes contradict humans and thus cannot be trusted. Therefore, this reliability issue belongs to the trustworthiness issues [144, 167]. The deviated behaviors studied in the second work refer to the situation where the DL applications produce different results given the same input before and after their deployment. This issue is closely related to the robustness of DL applications [196, 172] since the performance of these DL applications is not robust to their computational platforms. The last study of this thesis pinpointed that the baseline selection of DLGs is unfair. By revisiting the performance of DLGs using a fair baseline, we revealed that unfair baselines of DLGs are likely to result in overclaims of the effectiveness of DLGs. The unreliable assessment of DL applications discussed in this study provides a new perspective for software developers to understand the reliability of DL applications.

Chapter 3

Unreliable Inference within Deep Learning Applications

3.1 Introduction

Deep Learning (DL) have been widely deployed for image classification tasks [82, 160, 65, 67, 223]. While these DL-based classifiers outperform classic algorithms, such as SIFT+FV [152] and Sparse Coding [43], in terms of classification accuracy [82], which is the proportion of the inputs in test set whose inference result is the same as the ground truth, recent studies have raised concerns about other properties of such models, including reliability [144, 115, 163], fairness [178, 2, 209], robustness [22]. To help detecting the inappropriate behaviors of DNN models, various testing techniques [195, 42, 130, 172, 208, 45, 105] have been proposed. For instance, Pei *et al.* [130] proposed an optimization strategy to generate adversarial test inputs for image classification. Dwarakanath *et al.* [45] leveraged metamorphic testing to detect bugs in model implementations.

These techniques, however, do not consider a key property when evaluating a DL-based image classifier, that is, whether the inferences made by the model are based on the features encoded from the target objects or the features encoded from these objects' background. We refer to the former features as *object-relevant features*, the latter as *object-irrelevant features*, and the property as *object-relevancy property*. Intuitively, a reliable inference made by a DL-based image classifier should be mostly based on object-relevant features, instead of object-irrelevant features.

For instance, let us assume that the *mouse* shown in Figure 3.1a is the *target object*. The features encoded from it are object-relevant features, and the features encoded from

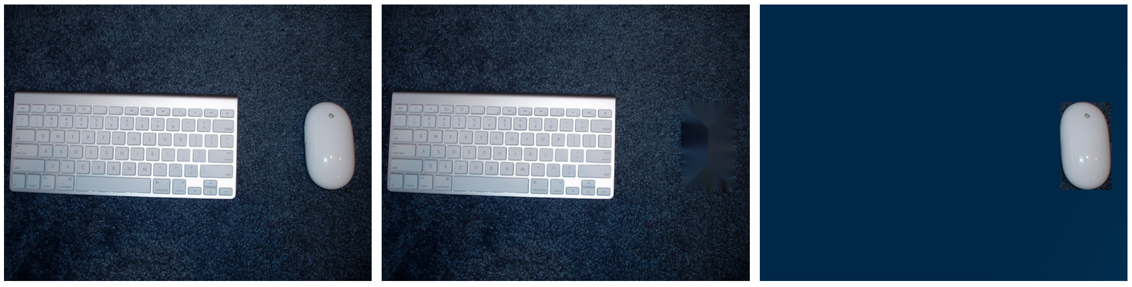

(a) (b) (c)

Figure 3.1: (a): The original image. (b): Object (mouse) corrupting mutation. (c): Object (mouse) preserving mutation.

the rest of this image are object-irrelevant. Let us further assume that a model classifies the image as shown in Figure 3.1a as “mouse”. This inference is reliable on the condition that it is made mostly based on the object-relevant features, instead of the object-irrelevant features. If the inference is majorly based on the object-irrelevant features but not the object-relevant features, the model is likely to classify the image in Figure 3.1b as “mouse” again, since this image has the same object-irrelevant features as Figure 3.1a. It is obvious that the image in Figure 3.1b does not have any “mouse” and should not be classified as “mouse”. Further, the model is also likely to classify the image as shown in Figure 3.1c as any label other than “mouse”, since this image does not have the object-irrelevant features in Figure 3.1a. It does not make sense since the image in Figure 3.1c clearly has the target object *mouse*.

Due to their stochastic nature, many DL-based image classifiers do not necessarily make inferences based on object-relevant features, which may lead to various problems. For instance, a recent study showed that an animal image classifier[1] would classify any image with bright backgrounds as “wolf”, regardless of the objects in the image [144]. This raises the concern of reliability and overfitting for this model [144, 106]. Another work showed that attackers could inject a backdoor trigger, such as a yellow square in an image’s background, to a deep neural network (DNN) model [56]. A model that makes inferences based on object-irrelevant features (e.g., yellow square at the background), will then classify an image containing this trigger to a specific label, regardless of the objects in the image. Thus, such models are not robust and can cause catastrophic consequences when being deployed in mission-critical applications. Based on the above analysis, we

[1]Unless otherwise specified, we use *classifier* to refer to *DL-based image classifier* in the remaining chapter.

conjecture that the violation of the object-relevancy property might be the root cause of many issues in DL-based image classifiers, including but not limited to the aforementioned ones. Therefore, it is important to develop effective techniques to assess the inference results of DL-based image classifiers from the perspective of object relevancy, so as to help improve the trustworthiness of DL-based image classifiers.

Validating the inference results of DL-based image classifiers with respect to object relevancy is challenging. It is well-known that DL models behave as black boxes [144, 130]. Their logic is learned from data and represented as model structures and weight values. It is non-trivial for human beings to examine the inference process of such models and check what kind of features determines the inference results. Some existing techniques [144, 154] try to explain the inferences for individual input. However, these techniques still require manual efforts to make the final assessment for each input due to the lack of test oracles. In contrast, in our work, we first try to generate both test inputs and test oracles for DL-based image classifiers, and then leverage them to identify unreliable inferences that violate the object-relevancy property automatically. However, generating test oracles is a long-standing challenge for software testing [12], especially in the testing of the deep learning systems [130, 172, 131, 121], where the expected probability outputted from DL models is unknown.

To tackle these challenges, we resort to metamorphic testing [24], which has been popularly leveraged to test DL-based image classifiers [195, 42, 208, 45]. Specifically, we propose two metamorphic relations (MRs) to quantitatively assess a model's inferences from the perspective of object relevancy as follows:

- **MR-1** An image mutated by corrupting only the features of the target object(s) should lead to an inference result with different label(s), or an inference result with the same label(s) but less certainty.
- **MR-2** An image mutated by preserving the features of the target object(s) and corrupting other features should lead to an inference result with the same label(s).

The two metamorphic relations will be formally defined in §3.3. For the purpose of metamorphic testing, we designed image mutation operations to generate test inputs with respect to the two relations. Applying these operations to a given image allows us to check if the pair of the original inference and the inference on a generated mutant satisfies the metamorphic relations. Violations of such relations will be deemed as the indication of unreliable inferences. We note that applying metamorphic testing to evaluate DL-based image classifiers is not new. However, existing work [172, 208] mutates the whole image (*e.g.* blurring or rotating) to test the model robustness. In comparison, our MRs focus on object-relevant/irrelevant features in one input image and hence our image mutation is regional, semantic and more targeted. Besides, our goal is to assess whether an inference violates the object-relevancy property, which is a new property proposed by us.

To validate the effectiveness of our proposed approach, we applied it to three popular DL-based image classifiers trained on the ImageNet dataset and one model trained on the COCO dataset [94], and then manually checked the results. The evaluation results show that for single-label classifiers, our approach achieves an aggregated precision of 64.1% for MR-1 and 96.4% for MR-2. As for multi-label classifiers, the corresponding precision for MR-1 and MR-2 is 78.2% and 86.5%, respectively. We also investigated the reasons for the false positives, and we found that they are mainly due to the inappropriate annotations of the dataset.

We then deployed our approach with the aim of investigating the pervasiveness of unreliable inferences. Specifically, we tested 18 pre-trained models for single-label classification from Keras [33] and 3 models for multi-label classification [65, 147]. We found that for each of them, more than thousands of correct classification inferences are actually unreliable, *i.e.*, they are not made based on object-relevant features. More seriously, we found that the pervasive unreliable inferences can cause significant bias on model evaluation. Specifically, our experiments revealed that unreliable inferences can cause significant degradation of a model's overall accuracy, thus preventing developers from correctly evaluating a model and fairly comparing among models. For example, after removing the unreliable inferences violating MR-2 in single-label image classification, the model accuracy is 8.84% higher than the original one. We also traced the ratio of unreliable inference during the model training and found that the current model training methodology is ineffective in terms of reducing unreliable inferences. Besides, enhancing a model with respect to its accuracy does not necessarily increase its probability to make reliable inferences. Therefore, developers need to design other methodologies with the aim to enhance a model's reliability, especially with respect to the object-relevancy property.

To summarize, we make the following contributions:

1. We proposed a metamorphic testing technique to automatically assess the reliability of inferences generated by DL-based image classifiers using object-relevant metamorphic relations.

2. We evaluated our technique and the results show that it is effective. Our approach can find thousands of unreliable inferences with high precision for each evaluated model.

3. We found that unreliable inferences are pervasive among a wide range of models. More seriously, such pervasive unreliable inferences significantly change the performance of DL-based image classifiers with respect to the accuracy, thus affecting model evaluation and comparison.

4. We explored the correlation between model accuracy and the ratio of unreliable inferences, and found that the current model training strategy should be further improved to help the model to learn the object-relevant features and avoid making unreliable inferences.

3.2 Preliminaries

3.2.1 Metamorphic Testing

Metamorphic testing [24, 25] was proposed to address the test oracle problem. It works in two steps. First, it constructs a new set of test inputs (called *follow-up inputs*) from a given set of test inputs (called *source inputs*) based on some properties that should be satisfied by the program under test. Second, it checks whether the program outputs based on the source inputs and the ones based on the follow-up inputs satisfy certain desirable properties, known as *metamorphic relations* (MRs).

For example, let us suppose p is a program implementing the $\sin()$ function. We know that the equation $\sin(\pi + x) = -\sin(x)$ holds for any numeric value x. Leveraging this knowledge, we can apply metamorphic testing to p as follows. Given a set of source inputs $I_s = \{i_1, i_2, \ldots, i_n\}$, we first construct a set of follow-up inputs $I_f = \{i'_1, i'_2, \ldots, i'_n\}$, where $i'_j = \pi + i_j$, $j \in [1, n]$. Then, we check whether the metamorphic relation $\forall j \in [1, n], p(i_j) = -p(i'_j)$ holds. A violation of it indicates the presence of faults in p.

3.2.2 Deep Learning-based Image Classification

Image classification is a key application of DL models. Its objective is to classify a given image into predefined labels. Popular DL models for image classification include AlexNet [82], VGG [160], ResNet [65], DenseNet [69], MobileNets [67] and so on. The performance of these models is mostly evaluated based on the top-1 accuracy, which refers to the percentage of test images whose correct labels are in the top-1 (sorted according to probability) inference made by models [82, 160, 65, 69, 67].

There are two types of image classification tasks, single-label classification and multi-label classification. In single-label classification, each input is supposed to be classified into one label. Figure 3.2a from ImageNet [39] shows an example input that is expected to be classified into label "tiger shark". Given an input i, the inference of a single-label classifier is a probability vector, $\mathbf{v}_i = [p_1, p_2, \ldots, p_n]$, where n is the number of labels. Each element p_j in the $\mathbf{v}_i$ represents the probability that the input belongs to the j-th label. The sum of the elements is equal to 1, *i.e.* $\sum_0^n p_j = 1$. The label with the highest probability is regarded as the final classification label of this classifier given this input. MNIST [89], CIFAR-10 [81], and ImageNet are common datasets for single-label classification.

In multi-label classification, the number of labels of each input is not limited to one. For example, Figure 3.2c from COCO [94] has three labels, {"person", "motorcycle", "airplane"}. In the classification, the inference result is regarded as correct if and only if it only contains the three labels [176, 190]. Similar to single-label classifiers, given an input i, the inference of a multi-label classifier is a probability vector, $\mathbf{v}_i = [p_1, p_2, \ldots, p_n]$, where n is the number of labels. Each element p_j in the $\mathbf{v}_i$ represents the probability that the input belongs to the j-th label. Unlike the single-label classifier, the sum of the elements is not necessarily equal to 1, *i.e.* $\sum_0^n p_j \neq 1$. The final classification result is the set of labels whose probability is equal to or larger than a predefined threshold, which is usually set to 0.5 [65, 147]. For example, given the input in Figure 3.2c, a multi-label classifier may output a probability vector $\mathbf{v}_i = [0.8, 0.7, 0.2, 0.6]$, where each element represents the probability of label "person", "airplane", "motorcycle" and "car", respectively. When the threshold is set to 0.5, the final classification result is {"person", "airplane", "car"}, which is an incorrect classification result as the "car" is not in the ground truth and the ground truth label "motorcycle" is not in the result. If the probability vector is $\mathbf{v}_i = [0.8, 0.7, 0.6, 0.2]$, the final result is {"person", "airplane", "motorcycle"}, and it is a correct classification result. Common multi-label datasets include COCO and Google Open Image [80].

3.3 Object-Relevant Metamorphic Relations

With the aim to identify the unreliable inference made by the DL-based image classifiers based on the object-irrelevant features, we are motivated to propose two metamorphic relations as mentioned in §3.1. This section presents the details of these two relations, starting with the motivating examples. Specifically, we follow a common metamorphic testing framework to define the two metamorphic relations [24, 25]. In subsequent formulation, let $\mathcal{M}(i)$ and $\mathcal{M}(i')$ denote the inferences made by a DL-based image classifier $\mathcal{M}$ on an input image i and its follow-up input i', respectively. Let $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'))$ denote

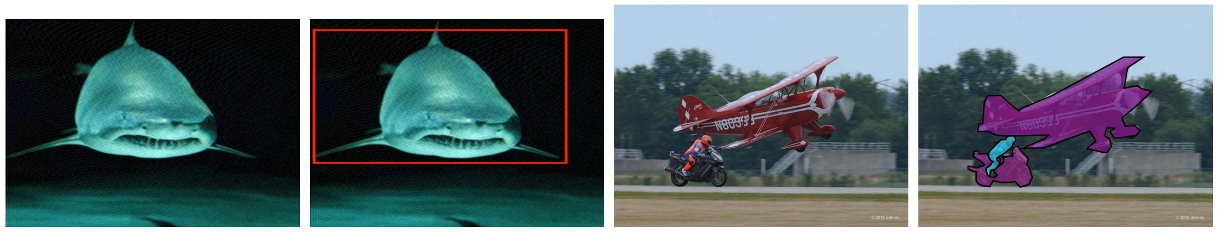

(a) (b) (c) (d)

Figure 3.2: Input examples and their annotations in image classifications. (a)(b): Image from the ImageNet dataset and its bounding box. Its label is “tiger shark”. (c)(d): Image from the COCO dataset and its object mask. Its labels are “person”, “motorcycle”, “airplane”.

the distance between two inferences $\mathcal{M}(i)$ and $\mathcal{M}(i')$.

3.3.1 Metamorphic Relation-1

Motivating Example-1 Given a source input as shown in Figure 3.1a, let us assume a DL-based image classifier predicts it as “mouse”. A follow-up input is constructed by corrupting the object *mouse*, as shown in Figure 3.1b. After feeding the follow-up input into the previous classifier, one of the following two cases could happen. First, it is possible that the label on follow-up input is still “mouse” and its certainty increases. Such a situation indicates that the inference on the source input is not based on the object(*mouse*)-relevant features. If it is based on the object(*mouse*)-relevant features, it does not make sense that the classifier still predicts it as “mouse” when there is no such object(*mouse*). This situation is out of human expectations on image classification, as humans will not classify the follow-up image that does not have *mouse* into label “mouse” with higher certainty. Second, it is possible that the inference on the follow-up input changes to another label, or the label remains the same but the certainty decreases. In other words, due to the corruption of the object(*mouse*)-relevant features, the classifier cannot make the inference with the same label and the same level of certainty as the one on source input. It implies that the inference on the source input is based on the object(*mouse*)-relevant features. This situation is in line with human expectations. Since the objects have been removed or corrupted, humans are likely to classify this image to a different label, or the same label but with less certainty. Motivated by the above example, we proposed the following MR-1. In the first situation aforementioned, the MR-1 is violated while in the second situation, MR-1 is satisfied.

MR-1 An image mutated by corrupting only the features of the target object(s) should lead to an inference result with different label(s), or an inference result with the same label(s) but less certainty.

Relation Formulation of MR-1 Let i'_c be a follow-up input constructed from a source input i for a DL-based image classifier $\mathcal{M}$ by corrupting the target object but preserving its background. We consider such a mutation as *object-corrupting*. An example of object-corrupting mutation is shown in Figure 3.1a (source input) and Figure 3.1b (follow-up input). MR-1 mandates that $\mathcal{M}(i)$ and $\mathcal{M}(i'_c)$ should satisfy the relation: $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) \geq \Delta_c$. Here D takes two factors of $\mathcal{M}(i)$ and $\mathcal{M}(i'_c)$ into consideration, *i.e.* the labels in the inferences and the certainty of the inferences. The detailed definition of D for MR-1 is introduced in §3.4.4. Δ_c denotes a threshold for the distance between two inference results made by a DL-based image classifier under metamorphic testing using object-corrupting mutations.

Explanation of MR-1 If an inference made by a specific DL-based image classifier is based on object-relevant features, after object-corrupting mutations, the new inference results should be affected since those object-relevant features have been corrupted, and thus those features cannot be further utilized by the classifier anymore. Such effects could cause two consequences. First, the classifier can still make the same inference as the inference of the original input while the certainty of the inference given by the classifier should be decreased since the object-relevant features have been corrupted. Second, the classifier cannot make the same inference as the inference of the original input if the corruption is very severe. Consequently, the label of the new inference should be different from the original one.

3.3.2 Metamorphic Relation-2

Motivating Example-2 Given a source input shown in Figure 3.1a, assume a DL-based image classifier predicts it as "mouse". A follow-up input is constructed by preserving the object, as shown in Figure 3.1c. After feeding the follow-up input into the previous classifier, one of the following two cases could happen. First, the inference on follow-up input is not "mouse" anymore. It indicates that the inference on the source input is not based on the object(*mouse*)-relevant features. Since the object *mouse* is still in the input, if the inference on the source input is based on the object(*mouse*)-relevant features, the inference should still be the "mouse". Second, the inference on follow-up input remains the same label. It implies that the inference on the source input is based on the object(*mouse*)-relevant features. When the object-relevant features are preserved, the classifier can leverage them to make the correct inference. Such a situation is in line with human expectations. Motivated by this example, we propose the following MR-2. In the above example, MR-2 is violated in the first situation and satisfied in the second situation.

MR-2 An image mutated by preserving the features of the target object(s) and corrupting other features should lead to an inference result with the same label(s).

Relation Formulation of MR-2 Let i'_p be a follow-up input constructed from a source input i for a DL-based image classifier $\mathcal{M}$ by preserving the target object(s) but mutating the other parts. We consider such a mutation *object-preserving*. An example of object-preserving mutation is shown in Figure 3.1a (source input) and Figure 3.1c (follow-up input). MR-2 mandates that $\mathcal{M}(i)$ and $\mathcal{M}(i'_p)$ should satisfy the relation: $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) \leq \Delta_p$. Here, Δ_p denotes a threshold for the distance between two inference results made by a classifier under metamorphic testing using object-preserving mutations. The detailed definition of D for MR-2 is introduced in §3.4.4.

Explanation of MR-2 If an inference made by a DL-based image specific classifier is based on object-relevant features, after object-preserving mutations, the labels of the new inference result should not be changed, since the object-relevant features are preserved and the classifier should be able to use them.

3.4 Approach

We present our approach in this section, starting from an overview of the whole approach, followed by the explanation of each stage.

3.4.1 Overview

Figure 3.3 shows the overview of our approach, including the following three stages:

① **Object-Relevant Feature Identification** Given an inference to be examined, we regard its input image as the source input. We semantically divide the input into two parts, a *target-object region* and a *background region*. The *target-object region* is where the target object(s) is located and where the object-relevant features are encoded. The *background region* is where the object-irrelevant features are encoded.

② **Follow-up Inputs Construction** Mutation functions are leveraged to generate follow-up inputs from the source inputs, based on the proposed metamorphic relations. Specifically, these mutation functions will corrupt, or preserve the object-relevant features

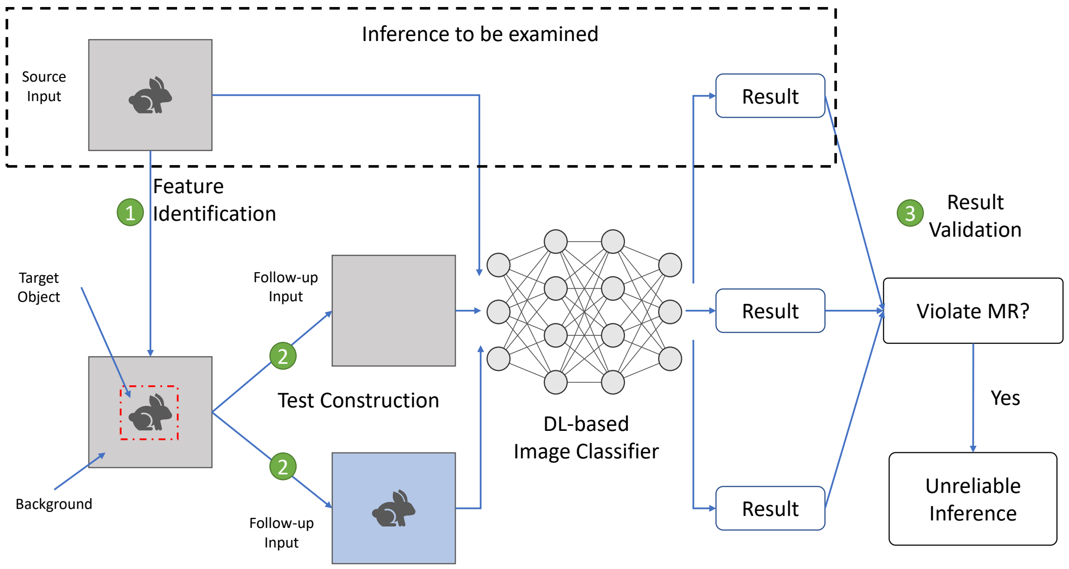


Figure 3.3: The overview of our metamorphic testing approach.

in the source input. The corresponding testing oracles will also be generated based on the metamorphic relations.

③ **Metamorphic Relation Validation** We validate if the distance between the inference result of a source input and the inferences of its follow-up inputs violates the test oracles. If so, the inference of the source input is flagged as an *unreliable inference*, which means this inference is made mainly based on object-irrelevant features.

Please note that our approach mainly assesses the correct inference results from DL-based image classifiers. In single-label classification, "correct" means that the top-1 label in the result is the same as the source input's ground truth. In multi-label classification, "correct" means that the set of labels in the results is the same as the set of labels in the source input's ground truth, as we mentioned in §3.2.2. We focus on correct inferences since if the inference result is incorrect, the target object might not exist in the input, and thus it is challenging to identify the object-relevant features.

3.4.2 Object-Relevant Feature Identification

In single-label classification, since each image only has a single label, we regard the object(s) belonging to the annotated label as the target object(s). For multi-label classification, each image can have multiple labels. We regard the union of all objects belonging to the annotated labels as the target objects. For example, for the input as shown in Figure 3.2c, the target objects consist of the airplane, motorcycle and person. In both cases, the pixels where the target object(s) reside are treated as the *target-object region* and the others are regarded as the *background region*.

The annotations of the target objects could be extracted from the dataset, obtained using object localization techniques, such as YOLO [142] and Faster R-CNN [143], or collected using recent image segmentation techniques like Segment Anything [78]. Currently, several datasets for image classification provide the annotation of objects, such as ImageNet, COCO, PASCAL VOC and Google Open Image. The annotations are usually in the format of a *bounding box*. For example, the bounding box of the *tiger shark* in Figure 3.2a is displayed as the red rectangle in Figure 3.2b. Some datasets, such as COCO, annotate the object using the object mask, which draws the boundary of each object with a finer granularity. These annotations provide the exact target-object region that does not contain any pixels belonging to the background region. Figure 3.2d shows the object marks of "person", "motorcycle" and "airplane".

Both annotation formats can be used in our approach. If the annotations are provided as bounding boxes, we regard the region of the bounding boxes as the target-object region. Although the target-object region could contain some pixels that do not belong to the target object(s), the majority of the region represents the target object. If the annotations are object masks, we regard the region covered by the object masks as the target-object region. In our experiment, we used the bounding box for the experiments based on the ImageNet dataset and the object mask on the COCO dataset, depending on the availability of the annotation format in these datasets.

3.4.3 Follow-up Inputs Construction

We generate the follow-up inputs by semantically corrupting or preserving the object-relevant features of a source image using the two aforementioned image mutations: *object-corrupting* mutation and *object-preserving* mutation.

There are many possible ways to design the mutation functions to corrupt or preserve the object-relevant features. However, it is challenging to quantitatively measure the degree of corruption and preservation. Such a challenge further brings difficulties to define the test oracle, as different levels of corruption and preservation should correspond to different designs of test oracle, especially the thresholds of test oracle (*e.g.* the Δ_c in §3.3). An inappropriate test oracle will influence the effectiveness of our approach.

To alleviate this challenge, we mutate the image by filling simple colors, such as white, gray and black, into the target-object region (or background region). Correspondingly, we use whether the classification results of source input and follow-up input are equal as the test oracle. The objective of our mutation is to simulate extreme cases, without considering the realism of images. For example, if the target-object region in the source input is substituted by black color, *i.e.* the object-relevant features are removed, but the classifier can still classify it correctly, the classifier is very likely to make the inference based on the object-irrelevant features. In real scenarios, our mutation can be considered as the simulation of the blocking of cameras. An existing study [130] designed for testing DL-based image classifier also generates test images via randomly patching black holes to images, in order to simulate the blocking of cameras.

Besides alleviating the above challenge, another advantage of using simple colors is that these colors bring little additional features to the source input. If we replace the object region with other objects or patterns, they may bring new features and further affect the classifier inference results. In such a situation, one cannot easily identify whether the change of the inference result is due to the absence of object-relevant features, or the appearance of these new features.

In our experiments, we use three colors, *i.e.* black (R0, G0, B0), gray (R127, G127, B127) and white (R255, G255, B255). For each source input, three follow-up inputs are generated based on MR-1 and three more are generated based on MR-2. For example, given the source input shown in Figure 3.4a, Figures 3.4b and 3.4c are two follow-up inputs generated based MR-1 and MR-2, respectively. It is possible that such simple colors could also induce bias to classifier inference. To alleviate this threat, eventually, we use the majority of their validation results as the final result. Such a strategy is called the *majority voting* [49] and it has been used by an existing study [130] to test DL systems. One threat to validity that might be raised is whether three colors are sufficient for performing metamorphic testing. To alleviate this threat, we compare the results using more colors in §3.5.2, and demonstrate that using three colors is sufficient.

Another threat that might be raised is why not using the inpainting technology to remove the object/background more naturally. Actually, we tried this method at the exploratory stage of this study. However, even the-state-of-art technology DeepFill [203] cannot completely remove the object features. An example is shown in Figure 3.4d. The

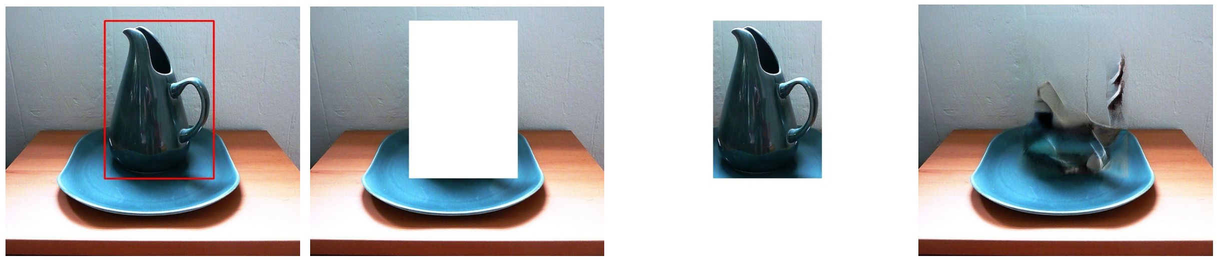


Figure 3.4: (a): Original image with bounding box. Its label is "pitcher, ewer". (b): Image after object corrupting mutation for MR-1. (c): Image after object preserving mutation for MR-2. (d): Image inpainting result using DeepFill.

feature of pitcher in the image cannot be removed completely. Moreover, such inpainting models usually need hundreds of hours for training and ∼15 seconds to inpaint an image, which is not efficient.

3.4.4 Metamorphic Relation Validation

In this subsection, we introduce the metamorphic relation validation process. Please note that in our experiments on single-label and multi-label classifiers, for each source input i, we generate three follow-up inputs i's. Then we will validate the MRs three times and use majority voting to decide whether MRs are violated. As we mentioned, such a method can mitigate the possible threat induced by a single mutation. We will regard $\mathcal{M}(i)$ as an unreliable inference if and only if MR-1 is violated at least two out of three times. The same strategy is applied for MR-2.

Validation of MR-1

MR-1 An image mutated by corrupting only the features of the target object(s) should lead to an inference result with different label(s), or an inference result with the same label(s) but less certainty.

Single-label Classification Here we use the same notation as §3.3. We define the distance function $\mathcal{D}$ as follows:

$$\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) = \begin{cases} 1, & \text{if } l_{\mathcal{M}(i)} \neq l_{\mathcal{M}(i'_c)} \\ & \text{or if } l_{\mathcal{M}(i)} = l_{\mathcal{M}(i'_c)} \text{ and } \mathcal{C}(l_{\mathcal{M}(i)}) > \mathcal{C}(l_{\mathcal{M}(i'_c)}) \\ 0, & \text{otherwise} \end{cases}$$

Here, $l_{\mathcal{M}(i)}$ is the label of the target object in $\mathcal{M}(i)$. $\mathcal{C}(\mathcal{M}(i))$ measures the certainty of the $M(i)$, according to the definition proposed by existing work in DL testing [197, 212]:[2]

$$\mathcal{C}(\mathcal{M}(i)) = \min_{0<j<n, j\neq l} |p_l - p_j|$$

where p_l is the probability of label $l_{\mathcal{M}(i)}$ and p_j is the probability of j-th label in the inference. Intuitively, the certainty measures the minimal difference between label $l_{\mathcal{M}(i)}$ and any other labels in terms of their probabilities. The value of $\mathcal{C}(\mathcal{M}(i))$ ranges in the region $[0, 1]$. The higher the value is, the more certain the DL-based image classifier is on the inference. If the inference is a correct inference, the above certainty equation actually calculates the difference between the highest probability and the second highest probability.

Correspondingly, we define Δ_c equals to 1. if $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) \geq \Delta_c = 1$, *i.e.* the label of the inference on the source input $l_{\mathcal{M}(i)}$ is different from the one of the inference on the follow-up input $l_{\mathcal{M}(i'_c)}$, or the labels are the same but the inference on the follow-up input become less certain, the MR-1 is satisfied. Otherwise, if $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) < \Delta_c = 1$, *i.e.* $l_{\mathcal{M}(i)}$ and $l_{\mathcal{M}(i'_c)}$ are the same and the certainty increases, it implies that after corrupting the object-relevant features in the source input, the DL-based image classifier can still correctly classify the input with more certainty. In other words, the examined inference $\mathcal{M}(i)$ is made based on features irrelevant to the objects. This conclusion violates our MR-1, and thus $\mathcal{M}(i)$ is labeled as an unreliable inference.

Multi-label Classification In multi-label classification, we adapt the above formula with slight modifications to cooperate with the multiple labels. Specifically, we use $L_{\mathcal{M}(i)}$ to denote the set of labels outputted by the DL-based image classifier $\mathcal{M}$ on input i. We define the distance function $\mathcal{D}$ as follows:

$$\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) = \begin{cases} 1, & \text{if } L_{\mathcal{M}(i)} \neq L_{\mathcal{M}(i'_c)} \\ & \text{or if } L_{\mathcal{M}(i)} = L_{\mathcal{M}(i'_c)} \text{ and } \mathcal{C}(L_{\mathcal{M}(i)}) > \mathcal{C}(L_{\mathcal{M}(i'_c)}) \\ 0, & \text{otherwise} \end{cases}$$

[2]The latter study refers this concept as “prediction confidence”.

To the best of our knowledge, the certainty in multi-label classification has not been defined by existing work, and the definition in single-label classification cannot be applied to multi-label classification directly. As we introduced in §3.2, in single-label classification, the sum of the probability of all labels is equal to 1. Labels are competing with each other and only the label with the highest probability is regarded as the final result. In other words, the increase of the probability of a label means the decrease of the probability of other labels. Thus, we can measure the certainty based on to what extent the probability of this label is different from the probabilities of the remaining labels. However, as we mentioned in §3.2.2, in multi-label classification, the probabilities of labels are relatively independent, *i.e.* the sum of the probabilities of all labels are not necessarily equal to 1. The difference between the probabilities of the two labels does not imply the inference certainty.

To address this challenge, in our approach, we regard the multi-label classification into multiple binary-classification tasks where each binary-classification predicts whether the input belongs to a single label or not. This enables us to measure the certainty of each label individually. For example, let us assume an inference result given by a multi-label classification is $[0.8, 0.9, 0.2]$, which corresponds to the probability of "airplane", "person" and "motorcycle". We can regard it as the outputs from three binary-classifiers. The first classifier predicts whether the input belongs to label "airplane" and outputs the probability 0.8. The second and third ones predict whether the input belongs to label "person" and "motorcycle", and output the probability 0.9 and 0.2, respectively. It is trivial to calculate the certainty of the binary classification task. Therefore, we can first measure the certainty of each binary classification, and then leverage the results to measure the certainty of multi-label classification.

More specifically, for any label l in the inference result of $\mathcal{M}(i)$ and its probability p, we define the certainty $\mathcal{C}_{l,\mathcal{M}(i)}$:

$$\mathcal{C}_{l,\mathcal{M}(i)} = |p - (1-p)| = |2p - 1|$$

The value of $\mathcal{C}_{l,\mathcal{M}(i)}$ is within the region $[0, 1]$. The intuition is to measure the certainty based on the difference between the probability that "it belongs to label l" and "it does not belong to label l". The larger the difference is, more certain the DL-based image classifier is on the inference. Based on the above definition of certainty of single label in the multiple-classification, we define the comparison of $\mathcal{C}(L_{\mathcal{M}(i)})$ and $\mathcal{C}(L_{\mathcal{M}(i'_c)})$ as following: $\mathcal{C}(L_{\mathcal{M}(i)}) > \mathcal{C}(L_{\mathcal{M}(i'_c)}) \iff \mathcal{C}_{l,\mathcal{M}(i)} > \mathcal{C}_{l,\mathcal{M}(i'_c)}, \forall\ l \in\ L_{\mathcal{M}(i)}$. The above equation compares the certainty of each label in the inferences on the source input and the follow-up input. Please note that for the predicate of certainty $\mathcal{C}(L_{\mathcal{M}(i)})$ and $\mathcal{C}(L_{\mathcal{M}(i'_c)})$, we check it only if the prior predicate $L_{\mathcal{M}(i)} = L_{\mathcal{M}(i'_c)}$ is true.

For Δ_c, we use the same definition as single-label classification, *i.e.* $\Delta_c = 1$. If $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_c)) \geq \Delta_c$, the MR-1 is satisfied. Otherwise, MR-1 is violated and the examined inference, *i.e.* $\mathcal{M}(i)$, is regarded as an unreliable inference.

Validation of MR-2

MR-2 An image mutated by preserving the features of the target object(s) and corrupting other features should lead to an inference result with the same label(s).

Single-label Classification We define $\mathcal{D}$ as follows:

$$\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) = \begin{cases} 0, & \text{if } l_{\mathcal{M}(i)} = l_{\mathcal{M}(i'_p)} \\ 1, & \text{otherwise} \end{cases}$$

Here, $l_{\mathcal{M}(i)}$ is the label with the highest probability in $\mathcal{M}(i)$. We define the threshold $\Delta_p = 0$. If $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) > \Delta_p = 0$, it means that the label of the inference on the source input $l_{\mathcal{M}(i)}$ is different from the one of the inference on the follow-up input $l_{\mathcal{M}(i'_p)}$. In other words, after preserving the features of the target object and corrupting the remaining features in the source input, the classifier classifies the follow-up input into a different label. This conclusion is opposite to our MR-2, and thus the examined inference $\mathcal{M}(i)$ is labeled as an unreliable inference. If $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) \leq \Delta_p = 0$, it implies that after preserving the features of the target object and corrupting the others, the classifier still classifies the input into the same label as the one of the source input. This result is in line with our MR-2 and thus the examined inference will not be labeled as an unreliable inference by us.

Multi-label Classification We define $\mathcal{D}$ as follows:

$$\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) = \begin{cases} 0, & \text{if } L_{\mathcal{M}(i)} = L_{\mathcal{M}(i'_p)} \\ 1, & \text{otherwise} \end{cases}$$

Here, $L_{\mathcal{M}(i)}$ is the set of labels in $\mathcal{M}(i)$. The equality of the $L_{\mathcal{M}(i)}$ and $L_{\mathcal{M}(i'_p)}$ is based on the equality of set. In other words, $L_{\mathcal{M}(i)} = L_{\mathcal{M}(i'_p)}$ if and only if for any element in $L_{\mathcal{M}(i)}$, this element is also in $L_{\mathcal{M}(i'_p)}$ and for any element in $L_{\mathcal{M}(i'_p)}$, it is also in $L_{\mathcal{M}(i)}$.

Same as single-label classification, the Δ_p is defined as 0. If $\mathcal{D}(\mathcal{M}(i),\ \mathcal{M}(i'_p)) > \Delta_p = 0$, it means $L_{\mathcal{M}(i)}$ and $L_{\mathcal{M}(i'_p)}$ are different. In other words, after preserving the features of the target object and corrupting the remaining features in the source input, the DL-based image classifier classifies the input into different labels with the inference on the source input. This conclusion is opposite to our MR-2, and thus the examined inference $\mathcal{M}(i)$ is labeled as an unreliable inference.

3.5 Evaluation

In this section, we evaluate our approach from the perspective of effectiveness. First, we investigate the effectiveness of our proposed approach to see whether it can successfully identify inferences that are made based on object-irrelevant features. Specifically, we measure the precision (true positive rate) of our approach, *i.e.* the number of real unreliable inferences in all inferences identified by our approach. We aim to answer the following question:

RQ1 What is the effectiveness of our approach in terms of true positive rate?

Further, as mentioned in §3.4, we use three colors to mutate inputs in our approach. One threat of our approach is that whether more colors should be used to identify unreliable inferences. To answer this question, we performed another experiment in which we use 15 distinct colors to mutate the inputs, and then compared it with the experiment in which only 3 colors are used. These results will help us to answer the following question:

RQ2 Is it sufficient to use only three colors for mutations in terms of effectiveness?

The source code and data of our experiment are available online at `https://github.com/yqtianust/PaperUnreliableInference`. Our experiments were conducted on two datasets, the ImageNet 2012 validation set and COCO 2014 validation set. The ImageNet 2012 validation set is a popular single-label classification dataset with 50,000 images. These images evenly distribute across 1,000 labels. The COCO 2014 validation set is a common multi-label classification dataset, with 40,504 images across 80 labels. On average, each image has 7.21 labels. We chose these datasets for three reasons. First, both are popular image classification datasets on which most state-of-the-art DL-based image classifiers are trained. Second, there are plenty of pre-trained DL-based image classifiers available as experiment subjects. Third, they provide the annotation of object boundaries.

3.5.1 Effectiveness of Our Approach

In order to evaluate whether our metamorphic testing approach can effectively identify unreliable inferences, we applied it to the inferences made by three pre-trained single-label

DL-based image classifiers from the Keras Application [32] and one multi-label classifiers [147]. The former ones are trained on the ImageNet dataset and the latter one is trained on the COCO dataset. Then we manually validated the testing results and measured the precision.

To validate whether the unreliable inferences identified by our approach are indeed made based on object-irrelevant features, for each of them, we manually checked the quality of their follow-up inputs. If the follow-up inputs are constructed as expected, *i.e.*, the object features in the follow-up inputs are corrupted(preserved) for MR-1(MR-2), we regarded the corresponding inference as indeed unreliable, *i.e.* a true positive case. If the follow-up inputs are not constructed as expected, the corresponding inference cannot be regarded as an unreliable inference, thus resulting in a false positive case.

More specifically, for the inference results that violate MR-1, we manually checked whether the object-relevant features were completely corrupted after the mutation, *i.e.*, whether the target objects in the follow-up inputs are indeed removed. If the follow-up input does not contain the target object, the inference violates MR-1 since the classifier still predicts it as the original label. Thus, this test result is a true positive. If the follow-up input still contains the target object, predicting it as the original label does not violate MR-1, and hence the identified unreliable inference is a false positive.

Similarly, for the inference that violated the relation MR-2, we manually checked whether the target objects were preserved and whether the other features were corrupted. Specifically, if the follow-up input contains the target object, the MR-2 is violated since the classifier does not predict the follow-up input as the original label. So, we labeled the test result as true positive. On the contrary, if the follow-up input does not contain the target object, MR-2 is not violated and the identified unreliable inference is a false positive.

The manual check was conducted by two graduate students individually and independently. Only the results agreed by consensus were considered. The disagreed results were labeled as “uncertain”.

Pilot Study

Before the manual check, we first conducted a pilot study to help us understand the possible cases (*i.e.* the root cause of false positive cases) that might be encountered in the manual check. Specifically, we randomly selected 200 unreliable inferences found by our approach to perform the pilot study, among which 100 violate MR-1 and the others violate MR-2. We investigated whether each unreliable inference is true positive and if not, what are the major reasons for those false positive cases.

In the investigation, for each unreliable inference, each student was requested to view a pair of inputs (pictures in our scenarios). More specifically, each pair of the inputs consisted of two inputs: (a) the source input on which the unreliable inference is made, *e.g.* Figure 3.4a, and (b) the follow-up input constructed based on the source input, *e.g.* Figure 3.4b if MR-1 is violated, or Figure 3.4c if MR-2 is violated. Besides, the label of the source input was provided to the students. The students were required to answer the following questions for the unreliable inferences violating MR-1:

1. Do you think the object-relevant features of the source input have been completely corrupted in the follow-up inputs, *i.e.* the target objects in the follow-up inputs have been indeed removed?

2. If not, please briefly explain the reason.

Similarly, for the unreliable inferences violating MR-2, the corresponding questions were:

1. Do you think the object-relevant features of the source input have been completely preserved in the follow-up inputs, *i.e.* the target objects still remain in the follow-up inputs?

2. If not, please briefly explain the reason.

First, two graduate students investigated the selected 200 image pairs individually and independently. Their answers to the questions have been recorded. Then for the inconsistent answers, they discussed with each other to see if they can reach a consensus. A reason is selected as a common reason if it occurs more than or equal to 10 times. Eventually, we finalized three common reasons inducing false positives for unreliable inference violating MR-1, which are:

(a) *Existence of Multiple Target Objects.* These false positives occurred because there are multiple target objects in the source input, but not all of them are corrupted in the follow-up inputs. Figure 3.5f shows an example. The original image in Figure 3.5b, whose label is "confectionery", has multiple confectioneries. Ideally, all of them should be corrupted in its follow-up inputs. However, after mutation, the follow-up input, as shown in Figure 3.5f, still contains multiple confectioneries since the dataset only annotates one of them, which is shown as the red rectangle in Figure 3.5b. As such, the inference of the follow-up input can still be "confectionery" as the object-relevant features (the other confectioneries) are not completely corrupted. Therefore, MR-1 is not violated and the original inference is a false positive of the identified unreliable inferences.

(b) *Incomplete Removal of the Target Object.* Some false positives occurred in the inputs that contain a single target object but only parts of it are corrupted in the follow-up input. An example is shown in Figure 3.5c, whose label is "drilling platform". Ideally, the entire platform should be corrupted in the follow-up inputs. However, the mutated images shown in Figure 3.5g still contain part of the target object. This is because the annotation provided by the ImageNet dataset does not cover the upper-half of "drilling platform", which differs from the other images in this label whose platforms are entirely annotated. Therefore, the follow-up input can lead to the same classification result as the original inference because the object-relevant features are not corrupted entirely. The MR-1 is not violated in this case.

(c) *Others.* It refers to the other reasons not belonging to the above two reasons. For example, the original image is not clear and hinders the students to identify the boundary of the target object.

For MR-2, we do not distinguish the reason for false positives since the number of false positives is very limited (less than 10 in our pilot study).

We also conducted a similar pilot study for multi-label classification. More specifically, we selected 50 unreliable inferences violating MR-1 and 50 ones violating MR-2 from all the unreliable inferences in multi-label classification found by our approach. A reason is considered common if it occurs at least 5 times. Since we did not notice other reasons than the ones aforementioned, we concluded the same reasons for both single-label and multi-label classifications.

Experiment Setup

DL-based Classifier Selection. For the single-label classifier, we selected NASNet-Large [223], MobileNet [67] and ResNet101 [65] among the pre-trained classifiers from the Keras Application [33] because their top-1 accuracy lies at the top, medium and bottom, respectively, among those of the classifiers. For the multi-label classifier, we selected TResNet-XL [147], since it achieves the highest accuracy on the COCO dataset to the best of our knowledge [147] till March 2021.

Sampling. We randomly sampled the inferences made by the four classifiers for the manual check, where the sample size is determined by the Cochran formula [35] with 95%

confidence level.

Manual Check. Two graduate students conducted the manual check similar to the pilot study. More specifically, each source input in the unreliable inference was displayed with the follow-up inputs constructed by our method. The students were asked the same question as the ones in the pilot study. The only difference is that at this time, the Q2 in unreliable inference violating MR-1 was supplied with three options, which are: (a)*Existence of Multiple Target Objects*, (b)*Incomplete Removal of The Target Object*, (c)*Others*. When (c) is chosen, the students were also required to write down detailed explanations. The students were allowed to choose multiple of the above options. During the manual check, we also monitored the reasons in (c) *Others*. If any reason in (c) *Others* occurs at least 10 times, we would extract a new common reason. Such a situation does not exist in our manual check.

Each student conducted the manual check individually and independently. It took around 15 hours for each of them to complete the manual check. After the individual check, they discussed the cases where the disagreement arises, in case any of them miss anything during the check. If the disagreement is addressed, the corresponding manual check result is changed. At last, we collected and analyzed the results. As we mentioned previously, only the results agreed by consensus were considered in the analysis. The Kappa Agreement Score [84] of the manual check is 0.955. Such a value indicates an almost perfect agreement between the two graduate students who conducted the manual check.

Threat to validity. There is a potential threat to validity in this experiment. Our manual check is subject to human mistakes. To address the threat, two graduate students conducted the manual check individually and independently. A result will be adopted only if both students made the same conclusion. The high Kappa Agreement Score indicates that the results is reliable.

Results and Discussion

Single-label Classifiers. Tables 3.1 and 3.2 show the manual check results for MR-1 and MR-2 for single-label classifiers, respectively. The column *Total* refers to the number of unreliable inferences identified by our approach for each classifier. Specifically, our approach identifies 1,392 inferences that violate MR-1 and 15,198 inferences that violate MR-2. We randomly sampled and manually checked 654 and 1,069 inferences from these two categories, respectively, as previously explained.

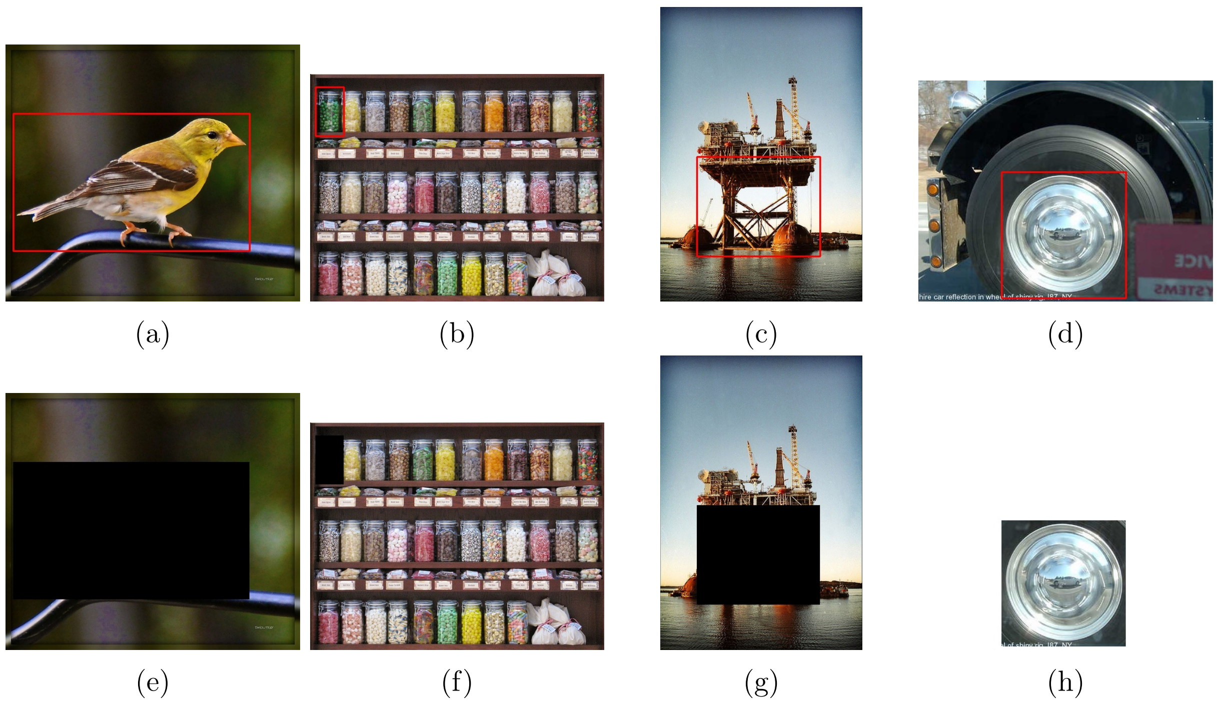


Figure 3.5: (a)(b)(c)(d): Images (with bounding boxes) as the source inputs. (e)(f)(g)(h): Images as the corresponding follow-up inputs. Labels: (a): "goldfinch, Carduelis carduelis", (b): "confectionery", (c): "drilling platform", (d): "car wheel".

Table 3.1: The manual check results for the effectiveness of MR-1 on single-label classifiers. Column *Multiple* is for the reason *existence of multiple target objects* and column *Incomplete* is for the reason *incomplete removal of the target object*. The number in the parentheses under *Multiple* is for cases shared by both reasons.

Classifiers	Accuracy	Total	Sample Size	True Positive	False Positive			Uncertain
					Multiple	Incomplete	Others	
NASNetLarge	82.7%	826	311	202 (65.0%)	84 (1)	16	4	6
ResNet101	76.4%	344	194	122 (62.9%)	55 (2)	8	8	3
MobileNet	70.3%	222	149	95 (63.8%)	42 (1)	8	3	2
Total		1,392	654	419 (64.1%)	181 (27.7%)	32 (4.9%)	15	11

Table 3.2: The manual check results for the effectiveness of MR-2 on single-label classifiers.

Classifiers	Accuracy	Total	Sample Size	True Positive	False Positive	Uncertain
NASNetLarge	82.7%	3,634	348	339 (97.4%)	1	8
ResNet101	76.4%	4,942	357	340 (95.2%)	7	10
MobileNet	70.3%	6,622	364	351 (96.4%)	0	16
Total		15,198	1,069	1,030 (96.4%)	8(0.07%)	34

As for the inputs that violate MR-1, the column *True Positive* of Table 3.1 shows that our approach achieves an average precision of 64.1%, ranging from 62.9% to 65.0% for different classifiers. Out of the 654 samples, 419 samples do not contain the target objects in the follow-up inputs but the classifiers keep labeling them as the target objects. So, they violate MR-1 and are true positive cases. Figure 3.5a shows an example, in which the original image is correctly classified by the classifier ResNet101 as "goldfinch, Carduelis carduelis". Although the follow-up input in Figure 3.5e does not contain birds, ResNet101 gives the same classification result as that of the original image, thus resulting in an unreliable inference.

We further checked the remaining 235 (=654 - 419) false positive cases, and found that 77.0% (=181/235) of the false positive cases are due to the *Existence of Multiple Target Objects* and 13.6%(=32/235) are because of *Incomplete Removal of the Target Object*. Moreover, there are four cases that belong to both *Existence of Multiple Target Objects* and *Incomplete Removal of the Target Object*. The above numbers (181 and 32) have included these four cases. Besides, there are 11 cases labeled as uncertain as the results from two students disagree with each other. The rest of the false positive cases (15 in total) are labeled as Others.

As for the inputs that violate MR-2, it shows that our approach achieves an aggregated precision of 96.4%, ranging from 95.2% to 97.4% for different classifiers. In total, 1,030 out of the 1,069 samples preserve the target objects in the follow-up inputs, but these follow-up inputs are not correctly classified by the classifiers. Therefore, these samples indeed violate MR-2 and they are regarded as true positives of the unreliable inferences violating MR-2. For the remaining 39 cases, only part of the target objects is preserved in the follow-up inputs. They do not violate MR-2 and are false positives. For instance, given the source input as shown in Figure 3.5d, the constructed follow-up input in Figure 3.5h

Table 3.3: The manual check results for the effectiveness of MR-1 and MR-2 on multi-label image classifier TResNet-XL.

MR	Total	Sample Size	True Positive	False Positive	Uncertain
MR-1	957	275	215 (78.2%)	44	16
MR-2	4,732	356	308 (86.5%)	30	18

only covers the center of wheel but not the entire tire. According to the definition from the WordNet [48] (the labels of the ImageNet dataset are defined according to WordNet), "car wheel" is *"a wheel that has a tire and rim and hubcap"*. Since the object-relevant features are only partially preserved, it makes sense that the follow-up input is incorrectly classified. Therefore, MR-2 is not violated and this is a false positive case.

We noticed that the precision of MR-2 is much higher than that of MR-1. We found the reason is that the aforementioned *Existence of Multiple Target Objects* will cause the follow-up input unqualified for the validation of MR-1, as the object-relevant features of the follow-up inputs will not be completely corrupted. However, such a situation will not affect MR-2 since as long as one of the target objects is preserved in the follow-up inputs, the follow-up inputs are valid to validate MR-2.

Multi-label Classifiers. Table 3.3 shows the manual check results for MR-1 and MR-2 for TResNet-XL, a multi-label classifier, respectively. The true positive rate for MR-1 and MR-2 is 78.2% and 86.5% respectively. This shows that our approach is also effective for multi-label classifiers. As for the false positives for MR-1, the major reasons are still *Existence of Multiple Target Objects* and *Incomplete Removal of The Target Object.* They account for 20 and 23 of the 44 false positive cases. The remaining one is due to the incorrect annotation, where a labeled broccoli is actually lettuce. For the false positives for MR-2, similar to single-label classification, the major reason is that their target objects are not completely preserved in the follow-up inputs and thus they do not violate MR-2.

Answer to RQ1: Our approach is effective in identifying unreliable inferences that violate MR-1 and MR-2, with an aggregated precision of at least 62.9% and 86.5%, respectively. The false positives are mainly caused by imperfect annotation of the target objects.

3.5.2 The Impact of The Number of Colors in Our Approach

As mentioned in §3.4, we use three colors to mutate inputs in our approach and use the majority of their results to identify the unreliable inference. One threat of our approach is that whether three colors are sufficient to identify unreliable inferences. To answer this question, we performed another experiment that uses 15 distinct colors to mutate the inputs, and we then compared the results obtained of the new experiment with that of the original one.

Experiment Design

Specifically, besides the three colors we used previously, we select 12 more commonly used colors, which are red (R255, G0, B0), maroon (R128, G0, B0), yellow (R255, G255, B0), olive (R128, G128, B0), lime (R0, G255, B0), green (R0, G128, B0), aqua (R0, G255, B255), teal (R0, G128, B128), blue (R0, G0, B255), navy (R0, G0, B128), fuchsia (R255, G0, B255), and purple (R128, G0, B128). We use the same approach as mentioned in §3.4. The only difference is that now we regard an inference as unreliable if and only if the MR is violated by at least 8 out of the 15 mutated inputs.

After the data collection, we compared the results using 15 colors and the ones using 3 colors. Statistically, we use the Chi-square independence test [50] to test the independence of the results obtained from the two approaches. The Chi-square independence test is commonly used to determine if there is a significant relationship between two categorical variables. In our experiment, we use it to determine if the decision "violate MR or not" using by three colors and the one using fifteen colors are strongly correlated. If yes, we can use three colors to save computation resources. We conduct the experiment using the pre-trained VGG16 from Keras.

Results and Discussion

We use variable V_3 to denote the decision "violate MR or not" according to the approach using three colors. Similarly, we use variable V_{15} to denote the decision "violate MR or not" according to the approach using 15 colors. We build the contingency tables for both MR-1 and MR-2 as shown in Table 3.4. The cell in the table represents the number of the inferences identified by the two approaches. For example, the cell "169" means there are 169 inferences that are considered as violating MR-1 by both the approach using three colors and the one using fifteen colors. The cell "1,467" means there are 1,467 inferences that

Table 3.4: Contingency tables for MR-1 and MR-2 to compare the experiment results obtained using 3 colors and 15 colors.

	MR-1		MR-2	
	V_3: Violate	V_3: Not Violate	V_3: Violate	V_3: Not Violate
V_{15}: Violate	169	52	5,145	1,467
V_{15}: Not Violate	63	35,323	249	28,773

are considered as not violating MR-2 by the approach using three colors and considered as violating MR-2 by the approach using the fifteen colors.

The p-values of the Chi-square test are both < 0.001 for MR-1 and MR-2, which is less than the typical threshold 0.05. The corresponding effect sizes[3] are 0.743 and 0.835 for MR-1 and MR-2, respectively. It indicates that the results obtained by the approach using three colors and the approach using fifteen colors are strongly correlated. In other words, if an inference is considered unreliable (or reliable) by the approach using three colors, the same decision will likely be made by the approach using fifteen colors, and vice versa. Overall, this experiment shows that using more colors than three in our approach has a minor difference compared to three colors. Therefore, it is sufficient to use three colors for the follow-up input construction in our approach.

Answer to RQ2: Using three colors in our approach is sufficient to identify unreliable inputs effectively.

3.6 Empirical Study

Leveraging our approach, we conduct an empirical study to understand the unreliable inference problems in reality. Specifically, we aim to answer the following research questions.

RQ3 How pervasive is unreliable inference in DL-based image classifier?

First, we want to understand the pervasiveness of the problem, *i.e.* to what extent are the inference results made by the state-of-the-art DL-based image classifiers based on object-irrelevant features. Specifically, we measure the proportion of unreliable inferences identified in all correct inferences outputted by these classifiers.

[3]in the Chi-square test, it is usually referred to as Cramér's V [36]

RQ4 Is there a correlation between the target object size and the unreliable inferences?

Second, we study the characteristics of the identified unreliable inferences. Specifically, we focus on the size of the target objects in unreliable inferences, a common attribute of objects. We studied whether there is any correlation between the object size and the unreliable inferences.

RQ5 To what extent will the unreliable inference affect the evaluation of image classifiers?

Next, we aim to understand the effect of such unreliable inferences. Specifically, we investigate whether the unreliable inferences can significantly affect the evaluation of image classifiers result, thus preventing us from correctly evaluating classifiers and comparing them fairly. In the experiments, we compare the accuracy of a classifier before and after removing those unreliable inferences from the associated test.

RQ6 Can the unreliable inference be tamed during training?

Finally, we investigate how to tame unreliable inferences. Specifically, we investigate whether the ratio of unreliable inferences can be reduced during the training process and whether it is correlated with the evaluation metrics such as accuracy. To achieve this goal, in the experiments, we track the ratio of unreliable inferences and the classification accuracy during the classifier training process .

3.6.1 Pervasiveness of Unreliable Inferences

RQ3 How pervasive is unreliable inference in DL-based image classifier?

Motivation

In the previous section, we showed that thousands of inferences made by the four pre-trained classifiers violate our MRs. In this subsection, we investigate the pervasiveness of the problem, *i.e.* whether such unreliable inferences generally exist in a wide variety of classifiers with different architectures. We leveraged our methodology to identify the unreliable inferences made by both the single-label and multi-label image classifiers. Then we measure the ratio of the unreliable inferences in all correct inferences. This research question can help us to understand the severity of the unreliable inferences.

Experiment Setup

We collected 21 pre-trained DL-based image classifiers from public repositories. 18 out of the 21 classifiers are single-label image classifiers, and they are collected from the Keras Application [33], a famous and popular repository for pretrained classifiers. All of them are trained on the ImageNet dataset, and their information (name and accuracy) is shown in the first two columns of Table 3.5. Besides the single-label classifiers, we also collected three multi-label classifiers, which are ResNet-50 [65], TResNet-L [147] and TResNet-XL [147]. ResNet-50 is chosen as it has been used as an experiment subject by existing papers [215, 176] and the other two classifiers are included because they are the state-of-the-art in terms of accuracy (till March 2021). All three multi-label classifiers are trained on the COCO dataset. Please note that the number of public available multi-label classifiers is much smaller than that of the single-label classifiers, and we have tried our best efforts to collect these three classifiers.

In the experiment, we found that Keras Application only provided the trained classifier, but missed the source code to reproduce the results for image classification, especially the code to preprocess the input. To avoid the possible mistakes in reproduction, we leveraged the functionality provided by an open-source toolbox, EvalDNN [174], which has successfully reproduced the reported accuracy for most of the 18 classifiers. The maximum difference between the reported accuracy and the reproduced one is only 0.7%, which demonstrates that we have faithfully deployed the classifiers in our experiments. For the multi-label classifiers, we successfully reproduced the results by leveraging the detailed source code provided by the authors [13] or related studies [177]. For the threshold in multi-label classifiers, we use the value suggested by their documentation, *i.e.* 0.5 for TResNet-L and TResNet-XL, and 0.7 for ResNet-50. The columns *Reproduced Accuracy* of Tables 3.5 and 3.6 list the accuracy reproduced in this study for single-label classification and multi-label classifiers, respectively. After the deployment, we applied our approach to identify unreliable inferences from all the correct inferences made by these classifiers.

Threats to validity. There are two potential threats to validity in this experiment. First, the classifiers used in this experiment may not include all the DNN-based image classifiers and our conclusion may have bias. To mitigate the threat, we collected 21 representative and popular classifiers. They covered most of the modern advanced architectures used in image classification. We believe that our conclusions can be generalized. Second, the inference results of these classifier can be affected due to the mistake in classifier deployments. To alleviate this threat, we leveraged the existing toolbox [174] and the source code provided by the authors. We ensured that the classifiers deployed in our experiment perform closely to the accuracy reported in their original research publications and documentations.

Results and Discussion

Tables 3.5 and 3.6 show the experimental results of single-label classifiers and multi-label classifiers, respectively. For each cell, the percentage displayed in the parentheses is the ratio of the number of unreliable inferences found by our approach with respect to the number of the correct inferences. Please note that the column *Inferences Violating MR-1* refers to the number of inferences violating MR-1, regardless of whether MR-2 is violated or not. The column *Inferences Violating MR-2* refers to the number of inferences violating MR-2, regardless of whether MR-1 is violated or not. The last column *Inferences Violating MR-1&2* refers to the number of inferences violating both MR-1 and MR-2.

The results reveal that each selected single-label and multi-label DNN classifier makes hundreds of unreliable inferences violating MR-1 and thousands of ones violating MR-2. In terms of ratio, for single-label classification, our approach identifies that 0.63%∼2.00% of the correct inferences violate MR-1, and 9.79%∼18.83% of the correct inferences violate MR-2. As for multi-label classification, the ratio is much higher. Specifically, our approach identifies that 4.71%∼5.49% of the correct inferences violate MR-1, and 24.38%∼34.91% of the correct inferences violate MR-2. Furthermore, there are around 2% of the inferences violating both MR-1 and MR-2. The results show that the phenomenon, *i.e.* classifier makes inferences based on object-irrelevant features, generally exists across different classifiers.

We further investigated whether different classifiers will make unreliable inferences towards different test inputs. If most of the classifiers make unreliable inferences for the same set of inputs, it is more likely that these inputs are defective. To conduct the investigation, we studied for each input the number of different classifiers whose inference for the input was unreliable. Specifically, the number of different classifiers varies from 1 to N, where N is the total number of classifiers included in our experiments. More specifically, N is 18 for single-label classification on the ImageNet dataset and 3 for multi-label classification on the COCO dataset. We then calculated the ratio of inputs, for which unreliable inferences were made by n classifiers ($n = 1, 2, ..., N$), with respect to the total number of inputs for which unreliable inferences were made by at least one classifier.

Figures 3.6a and 3.6b show the results for single-label DL classifiers on the ImageNet dataset and multi-label classifiers on the COCO dataset, respectively. It can be observed that, for single-label classification, 43.7% and 31.8% of the inputs concern unreliable inferences violating MR-1 and MR-2 made by only one classifier, respectively. More than half of the inputs concern unreliable inferences made by three or fewer classifiers. Only

Table 3.5: The number and ratio of unreliable inferences in single-label image classifiers on the ImageNet dataset.

Classifiers	Reproduced Accuracy	Inferences Violating MR-1	Inferences Violating MR-2	Inferences Violating MR-1&2
Xception	79.0%	374 (0.95%)	4,104 (10.39%)	229 (0.58%)
VGG16	71.3%	259 (0.73%)	5,394 (15.14%)	228 (0.64%)
VGG19	71.3%	252 (0.71%)	5,628 (15.80%)	219 (0.61%)
ResNet50	74.9%	253 (0.68%)	5,248 (14.01%)	197 (0.53%)
ResNet101	76.4%	344 (0.90%)	4,942 (12.93%)	268 (0.70%)
ResNet152	76.6%	334 (0.87%)	4,727 (12.34%)	266 (0.69%)
ResNet50V2	75.3%	247 (0.66%)	5,387 (14.30%)	213 (0.57%)
ResNet101V2	76.9%	271 (0.70%)	4,606 (11.98%)	212 (0.55%)
ResNet152V2	77.7%	319 (0.82%)	4,392 (11.30%)	252 (0.65%)
InceptionV3	77.9%	404 (1.04%)	4,663 (11.98%)	292 (0.75%)
InceptionResNetV2	80.4%	686 (1.71%)	3,998 (9.94%)	388 (0.97%)
MobileNet	70.3%	222 (0.63%)	6,622 (18.83%)	195 (0.55%)
MobileNetV2	71.2%	281 (0.79%)	6,437 (18.08%)	225 (0.63%)
DenseNet121	75.0%	278 (0.74%)	4,349 (11.60%)	219 (0.58%)
DenseNet169	76.2%	340 (0.89%)	4,154 (10.91%)	264 (0.69%)
DenseNet201	77.3%	334 (0.86%)	4,296 (11.11%)	260 (0.67%)
NASNetMobile	73.8%	461 (1.25%)	6,505 (17.64%)	345 (0.94%)
NASNetLarge	82.7%	826 (2.00%)	3,634 (8.79%)	383 (0.93%)

Table 3.6: The number and ratio of unreliable inferences in multi-Label image classifiers on the COCO dataset.

Classifier	Reproduced Accuracy	Inferences Violating MR-1	Inferences Violating MR-2	Inferences Violating MR-1&2
ResNet50	34.5%	657 (4.71%)	4,873 (34.91%)	422 (3.0%)
TResNet-L	45.5%	1,013 (5.49%)	5,028 (27.26%)	442 (2.4%)
TResNet-XL	47.9%	957 (4.93%)	4,732 (24.38%)	362 (1.9%)

a small portion of inputs (less than 2.7%) concern unreliable inferences made by all 18 classifiers. A similar pattern can also be found for multi-label classifiers. 74.3% and 67.4% of the inputs concern unreliable inferences violating MR-1 and MR-2 made by only one classifier, respectively. Less than 7.8% of the concern unreliable inferences made by all three classifiers.

Such results reveal that different classifiers make unreliable inferences for different sets of inputs, which indicates that such unreliable inferences are more likely to be caused by the classifiers themselves instead of the inputs.

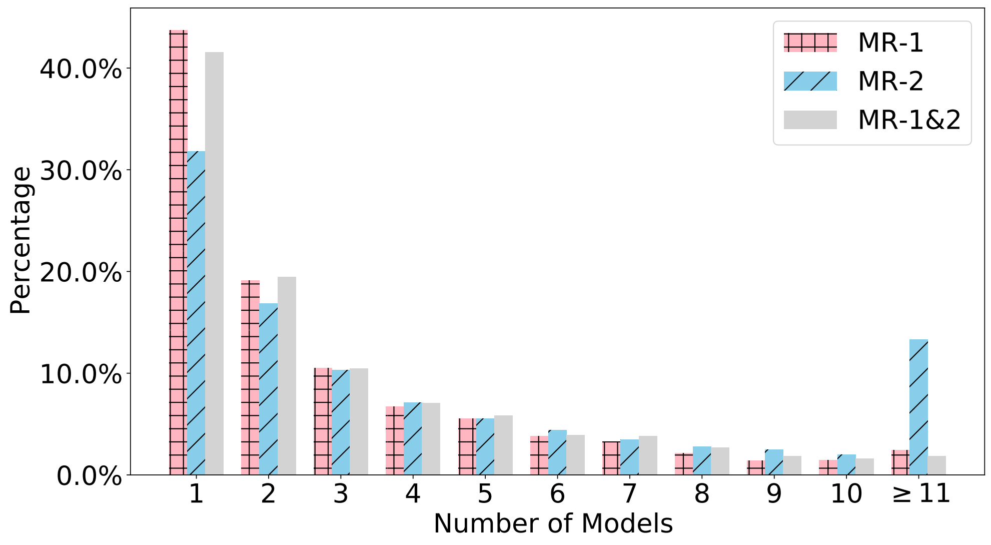
Answer to RQ3: The problem of making unreliable inferences is common to state-of-the-art DL-based classifiers. Since these classifiers make unreliable inferences on different input sets, the problem is likely to be caused by classifiers instead of inputs.

3.6.2 Characteristic of Unreliable Inferences

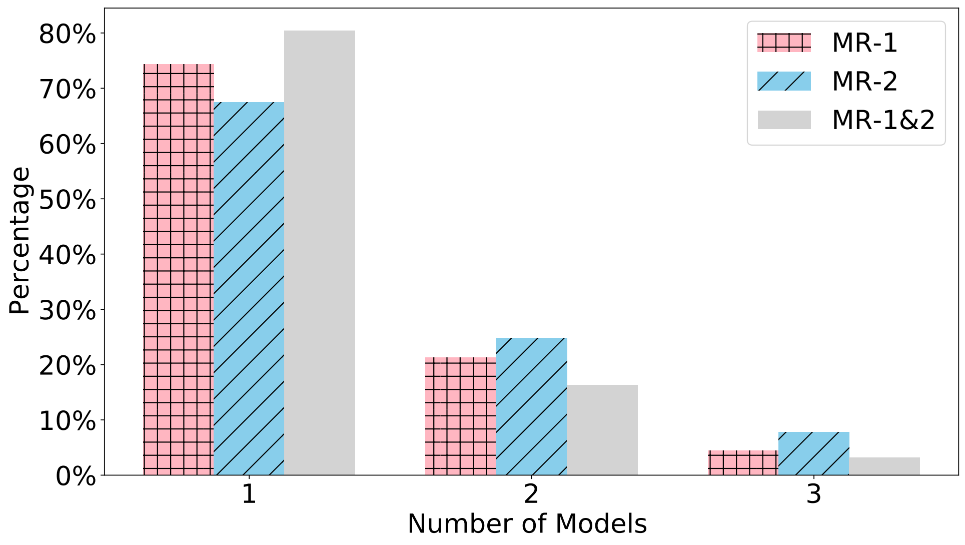
RQ4 Is there a correlation between the target object size and the unreliable inferences?

Motivation

As shown in §3.6.1, unreliable inferences are pervasive, and different classifiers make unreliable inferences for different inputs. We are curious about whether common characteristics exhibit among unreliable inferences. If so, we may give some useful suggestions to developers.



(a) Single-label DL-based classifiers on the ImageNet dataset



(b) Muti-label DL-based classifiers on the COCO dataset

Figure 3.6: The percentage of inputs for which unreliable inferences were made by different number of single-label and muti-label Classifiers.

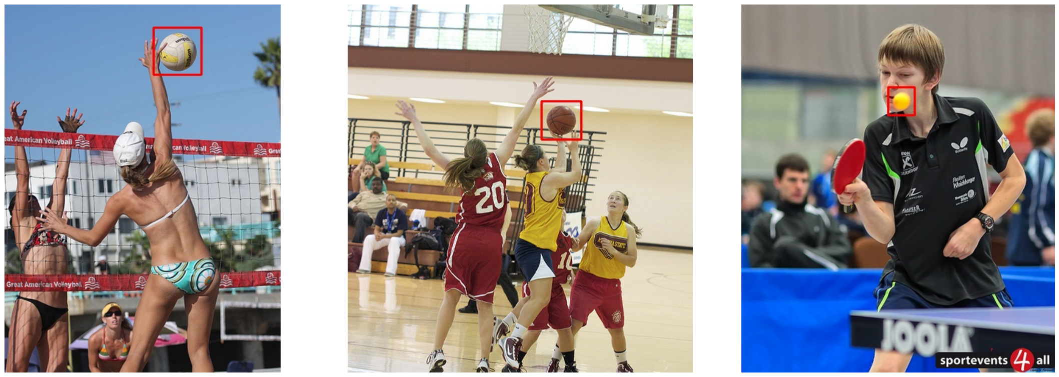


Figure 3.7: The images with small objects are unreliably inferred by DL-based image classifiers.

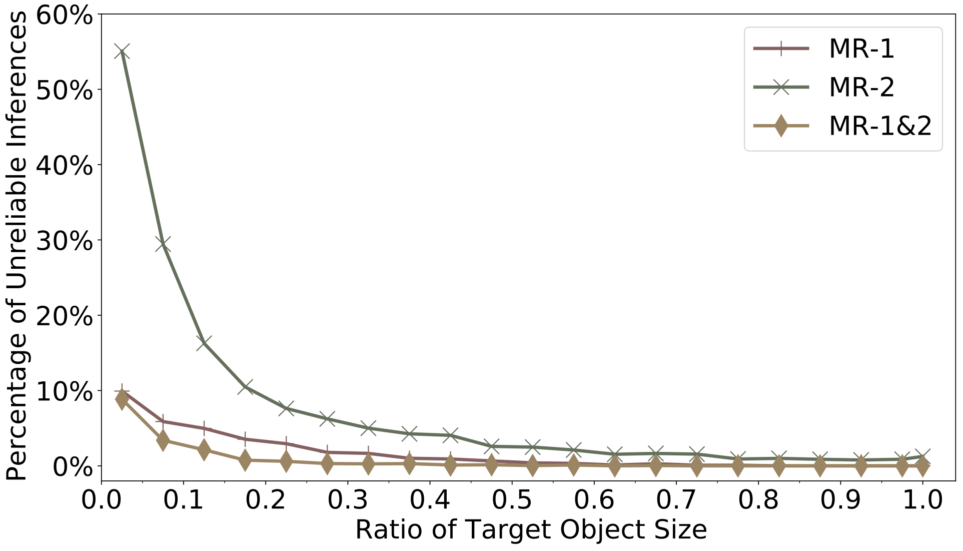
We manually investigated those inputs that cause unreliable inferences made by most classifiers. We observed that the sizes of the target objects in these inputs usually occupy a tiny part of the whole image. Figure 3.7 shows some examples. The objects of these images are different types of balls whose sizes are often small in the images, especially compared to the sports facilities and players. It motivates us to investigate whether the size of an input's target object is correlated with its probability of being unreliably inferred by DL-based image classifiers.

Experiment Design

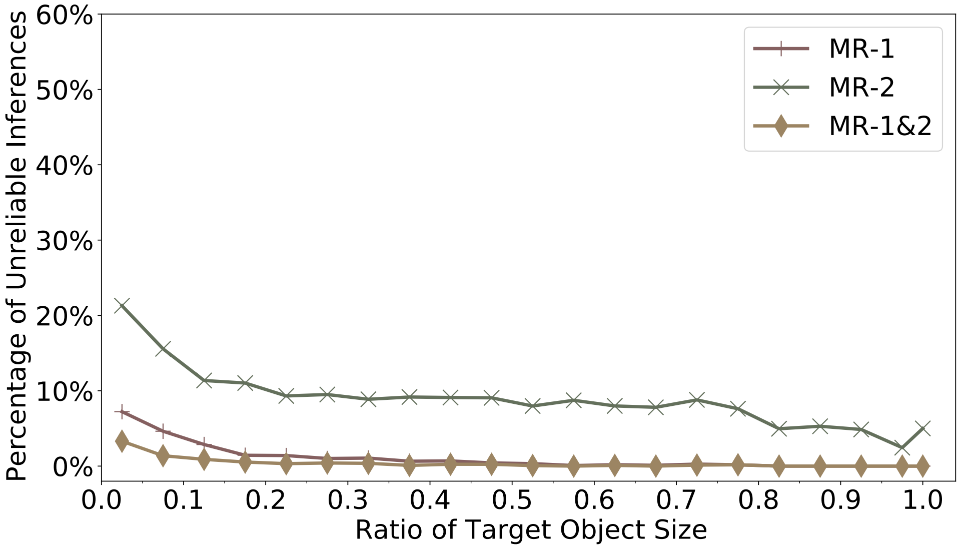
To answer this question, for each unreliable inference, we computed the ratio of the target object's size with respect to the size of the whole input image. Then we divided all inferences into 20 intervals based on their ratios, which are $[0.05 * i, 0.05 * (i + 1))$ and i ranges from 0 to 20. For each interval, we computed the ratio of unreliable inferences, with respect to the total number of inferences belonging to this interval. We selected the NASNetLarge and TResNet-XL as experiment subjects since they achieved the highest top-1 accuracy in all classifiers used in our experiment for the ImageNet dataset and the COCO dataset, respectively.

Results and Discussion

Figures 3.8a and 3.8b show the results for the classifier NASNetLarge on the ImageNet dataset and for the classifier TResNet-XL on the COCO dataset, respectively. Please



(a) Single-label classifier NASNetLarge on the ImageNet dataset.



(b) Multi-label classifier TResNet-XL on the COCO dataset.

Figure 3.8: The ratio of unreliable inferences made by single-label and multi-label classifiers *w.r.t* the ratio of target object size.

note that for each interval, we use its middle point as the value in x-axis, except for the last interval we use the point 1.0. We observed that for these inferences whose target objects are smaller (relative to the size of the image), they are more likely to be unreliable. Similar results have been observed among the other classifiers. We suspect that when the classifier handles an image whose target object is small, it often extracts features from the background region. Eventually, it leverages object-irrelevant features to make decisions.

Answer to RQ4: In summary, we found that inputs with small target object sizes are more likely to be unreliably inferred by existing DL-based image classifiers. We suggest the users of these classifiers to pay more attention when making inferences on these inputs (*i.e.* the objects' size are less than 30% of the whole image), especially when deploying these classifiers on safety-critical applications.

3.6.3 Effect of Unreliable Inferences

RQ5 To what extent will the unreliable inference affect the evaluation of image classifiers?

Motivation

As revealed by previous sections, a significant proportion of the correct inferences made by existing classifiers are unreliable. Such pervasiveness of unreliable inferences might cause bias in understanding and evaluating the performance of different classifiers. Specifically, if there exists a significant amount of unreliable inferences, it could induce non-trivial uncertainties in measuring the accuracy of classifiers. Therefore, we investigated the effect of unreliable inferences on the accuracy evaluation of classifiers in this experiment.

Experiment Design

We investigated the effects of unreliable inferences on the measurement of accuracy. Since both the correct and incorrect inferences can be unreliable and both of them are important to classifier evaluations, in this section, we examined both correct and incorrect inferences. For the incorrect inferences, it is possible that they have the labels that do not exist in the ground truth and thus the object-relevant features cannot be directly identified. In such cases, we use the union of all the objects in the annotation to approximate the target object and then identify the object-relevant features.

In the investigation, with respect to MR-1, we examined all (both correct and incorrect) inferences and separated them into two sets for each classifier according to whether they are reliable. One set contains all the inputs whose inferences are identified as unreliable by our approach and another set that contains the remaining test inputs. We denoted the former set as “Unreliable” and denoted the latter one as “Reliable”. We also compared the results such obtained with the original accuracy reproduced by our approach, which is denoted as “Original”. Similar procedures were applied with respect to MR-2, and the MR-1&2. If the results before and after removing the unreliable inference have a significant difference, it indicates that the unreliable inferences will induce bias for classifier evaluation. We then re-computed the accuracy based on each set of test inputs and checked if the evaluation results are significantly different by conducting the Wilcoxon signed-rank test [189].

Results and Discussion

Table 3.7 shows the results aggregated over all the 18 single-label image classifiers. In terms of the accuracy evaluated after removing the unreliable inferences with respect to MR-2 (column *MR-2 Reliable*), it is significantly higher than the original accuracy value obtained over all the test inputs (p-value $= 3.81 * e^{-6}$). On average, the classification accuracy after removing the unreliable inferences is 8.84% (5.73%∼12.31%) higher than the original accuracy. For MR-1 and MR-1&2, a certain trend toward significance could also be observed, for which the classification accuracy after removing the unreliable inferences is only 1.31% (1.04%∼1.55%) and 1.30% (1.04%∼1.52%) higher than the original accuracy.

Table 3.8 shows the result of three multi-label classifiers. Similar to the previous finding for single-label classifiers, after removing the unreliable inferences violating MR-2, the classification accuracy is much higher (13.79%∼26.95%) than the original accuracy value obtained over all the test inputs. Please note that the significant test is not applicable since there are only three samples, which is significantly less than 20, the typical minimum number for a significant test.

The above results reveal that the existence of unreliable inferences violating MR-2 causes significant bias for classifier evaluation, while the effect of unreliable inferences violating MR-1 and MR-1&2 is limited. By excluding those unreliable inferences violating MR-2, the performance of existing classifiers evaluated with respect to accuracy is much higher than that evaluated based on inputs containing unreliable inferences. We suggest developers to remove unreliable inferences for fair classifier comparisons, especially the inferences violating MR-2.

Besides, in general, as shown in Tables 3.7 and 3.8, the classification accuracy on the

Table 3.7: The comparison of the top-1 accuracy between the unreliable inferences and reliable inferences for single-label image classifiers

Classifier	Original	MR-1		MR-2		MR-1&2	
		Unreliable	Reliable	Unreliable	Reliable	Unreliable	Reliable
Xception	79.02%	28.22%	80.42%	45.51%	86.40%	20.00%	80.41%
VGG16	71.27%	21.73%	72.48%	41.10%	82.01%	20.07%	72.46%
VGG19	71.26%	20.62%	72.52%	42.21%	81.82%	18.83%	72.50%
ResNet50	74.93%	21.58%	76.21%	44.98%	84.05%	18.17%	76.19%
ResNet101	76.42%	26.34%	77.76%	45.48%	85.01%	22.48%	77.74%
ResNet152	76.60%	26.13%	77.94%	44.88%	85.07%	22.52%	77.91%
ResNet50V2	75.34%	19.92%	76.78%	46.19%	84.21%	17.82%	76.75%
ResNet101V2	76.89%	21.16%	78.37%	45.05%	85.08%	17.76%	78.34%
ResNet152V2	77.73%	23.19%	79.28%	45.51%	85.44%	19.73%	79.25%
InceptionV3	77.87%	30.01%	79.20%	46.05%	85.95%	24.48%	79.17%
InceptionResNetV2	80.41%	42.52%	81.68%	47.43%	87.10%	30.36%	81.73%
MobileNet	70.34%	21.50%	71.38%	42.84%	82.65%	19.52%	71.37%
MobileNetV2	71.19%	23.44%	72.37%	44.47%	82.08%	20.09%	72.36%
DenseNet121	74.97%	22.32%	76.34%	41.89%	83.64%	18.73%	76.32%
DenseNet169	76.18%	26.19%	77.51%	42.02%	84.59%	22.30%	77.48%
DenseNet201	77.32%	26.05%	78.67%	44.60%	85.13%	22.34%	78.63%
NASNetMobile	73.77%	33.12%	74.94%	46.18%	84.60%	27.38%	74.97%
NASNetLarge	82.68%	46.74%	83.99%	49.44%	88.40%	30.25%	84.04%

Table 3.8: The comparison of the top-1 accuracy between the unreliable inferences and reliable inferences for multi-label image classifiers.

Classifier	Original	MR-1		MR-2		MR-1&2	
		Unreliable	Reliable	Unreliable	Reliable	Unreliable	Reliable
ResNet50	34.5%	41.66%	34.17%	22.47%	48.29%	33.65%	34.49%
TResNet-L	45.5%	45.98%	45.51%	22.87%	72.45%	28.85%	46.19%
TResNet-XL	47.9%	45.31%	48.06%	23.19%	73.03%	25.84%	48.71%

unreliable inference is significantly lower than the original accuracy of classifier. However, there are some exceptions. In multi-label image classification (Table 3.8), the classification accuracy on the unreliable inference is higher than (ResNet50 and TResNet-L, MR-1) or close to (ResNet50, MR-1&2 and TResNet-XL, MR-1) the original accuracy of the classifier. We suggest that the developers should pay more attention to such exceptions: even if the unreliable inferences have a comparable accuracy with the reliable ones, they may raise concerns on classifier reliability, as we mentioned in §3.1.

Answer to RQ5: The unreliable inferences violating MR-2 can cause significant effects (8.84% for single-label classification and 21.96% for multi-label classification) on the evaluation results, thus inducing bias in classifier comparisons. On the contrary, the effect of the unreliable inferences violating MR-1 and MR-1&2 is limited.

3.6.4 Taming Unreliable Inferences

RQ6 Can the unreliable inference be tamed during training?

Motivation

Previous results have shown that unreliable inferences generally exist in widely-used classifiers built with different architectures. Besides, the inputs causing unreliable inferences vary across classifiers. These unreliable inferences can induce significant bias in the evaluation of classifier performance. In this subsection, we studied whether such unreliable inferences can be tamed. Specifically, our study has two goals.

First, we investigated whether the ratio of unreliable inferences can be reduced during the classifier training process. Second, we investigated whether there is any correlation between classification accuracy and the ratio of unreliable inferences. Understanding their correlation helps formulate a training strategy taming such unreliable inferences. For instance, if the top-1 accuracy is negatively correlated with the ratio of unreliable inference, the ratio of the unreliable inferences is likely to be reduced by enhancing the classification accuracy.

Experiment Setup

We conducted two experiments with the aim to achieve the above two goals. First, we trained the VGG16 and Resnet50 classifiers from scratch using the training source code provided by PyTorch official example repository [134] based on the ImageNet dataset. We selected these two classifiers because they have been popularly adopted by existing studies for testing DNN systems [130, 104, 176, 215]. The training was based on the default hyper-parameters, and stopped when its accuracy and loss reach saturation. We then measured the ratio of unreliable inferences in all correct inferences for every five epochs during the training process to see if they are reduced. Since the training process of DL-based image classifiers is stochastic, we repeated the training three times for each classifier. Please note that the training of these two classifiers is very time-consuming. Although our server has eight NVIDIA 2080Ti GPU cards, it still takes around 80 mins and 30 mins to train one epoch for VGG16 and Resnet50. The total training time spent for this experiment is more than 20 days.

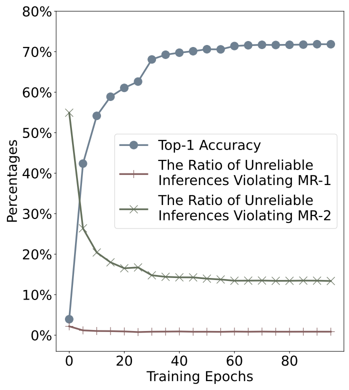
Second, we investigated the correlation between classification accuracy and the ratio of unreliable inferences using the pre-trained classifiers in Table 3.5. Specifically, we used the Pearson Correlation [14] to check whether the ratio of unreliable inferences and the top-1 accuracy are correlated. We also plotted them for visualization.

In this research question, we did not include the multi-label classification due to the following two reasons. First, the source code to train these classifiers is not available. Second, the number of available multi-label classifiers is limited and it is not applicable to calculate the Pearson Correlation.

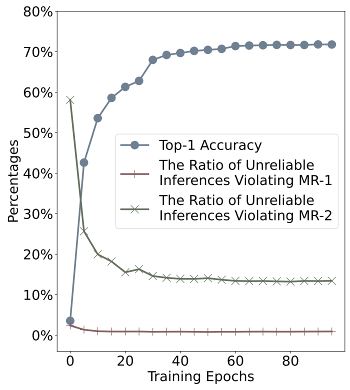
Results and Discussion

On average, our trained VGG16 and Resnet50 classifiers achieve the top-1 accuracy of 72.1% and 76.1%, respectively. Their accuracy is close to the accuracy of the pre-trained classifiers published by PyTorch [135], which are 71.6% and 76.2%, respectively. Figure 3.9 shows the top-1 accuracy and the ratio of unreliable inferences during the training stages. Please note the ratios of unreliable inferences violating MR-1&2 are not plotted as they are highly overlapped with the ratios of unreliable inferences violating MR-1.

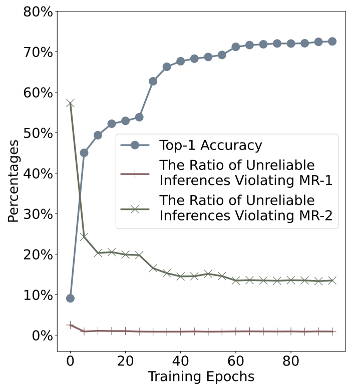
It can be observed that at the beginning of training, the ratio of unreliable inferences violating MR-2 decreases significantly and the ratio of unreliable inferences violating MR-1 slightly decreases. Later on, both of them become stable with the accuracy becoming saturated. Such results indicate that the current classifier training methodologies can guide the classifiers to learn object-relevant features to certain extents, as the ratio of unreliable inferences decreases at the first beginning. However, they become less effective with the training epochs increases, as the ratio of unreliable inferences becomes stable after the



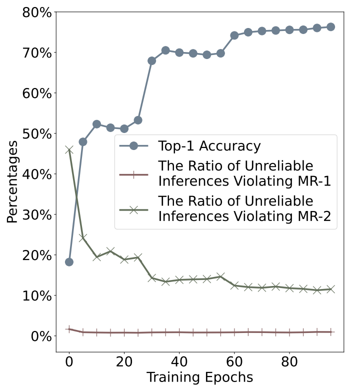
(a) VGG16, Seed 1.



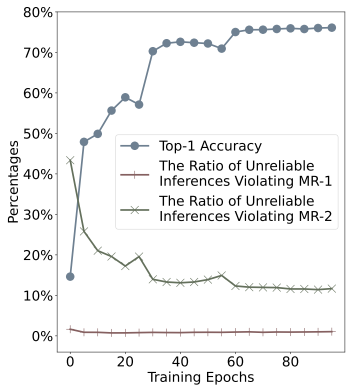
(b) VGG16, Seed 2.



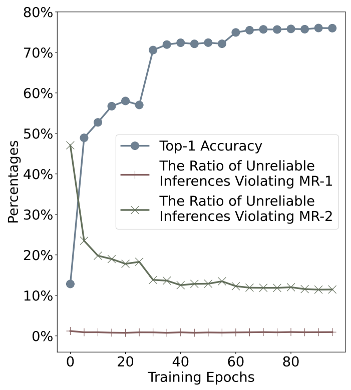
(c) VGG16, Seed 3.



(d) Resnet50, Seed 1.



(e) Resnet50, Seed 2.



(f) Resnet50, Seed 3.

Figure 3.9: The top-1 accuracy and the ratio of unreliable inferences of VGG and Resnet50 during training. Training is repeated three times using different seeds.

beginning. In other words, they may not necessarily prevent the classifier from making unreliable inferences.

We then investigated the correlation between the top-1 accuracy and the ratio of unreliable inferences based on the pre-trained classifiers. The Pearson Correlation coefficients between the ratio of unreliable inferences violating MR-1, MR-2, and MR-1&2 with top-1 accuracy are 0.702, -0.901, and 0.492, respectively. Figure 3.10 shows the relation of the ratio of unreliable inferences that violate MR and the top-1 accuracy, as well as their linear regression lines. The results indicate a strong negative correlation (-0.901 < -0.9) between the ratio of unreliable inferences violating MR-2 and top-1 accuracy. In other words, higher top-1 accuracy of a classifier couples with lower ratio of its unreliable inferences violating MR-2. The ratio of unreliable inferences violating MR-1 has a relatively positive correlation with top-1 accuracy. It increases very slightly with the increase in top-1 accuracy. The ratio of unreliable inferences violating MR-1&2 remains about the same. This may be because that the ratio of unreliable inference violating MR-1 and MR-1&2 is relatively small and their changes are not obvious.

Answer to RQ6: The current training methodologies can help the classifiers to reduce the unreliable inference to certain extents, but they become less effective with the training epochs increases and may not necessarily prevent the classifier from making unreliable inferences.

3.7 Limitation and Future Work

Our study points out that unreliable inferences commonly exist in the DL-based image classifiers. In this section, we discuss some limitations of our work and the future work. In the future, we will explore the possibility to improve the reliability of inferences made by DNN classifiers and address such unreliable inferences effectively and efficiently.

3.7.1 Other Possible Metamorphic Relations

We introduced our approach for the MR-1 and MR-2 in §3.4. There are alternative approaches. For example, in the multi-label classification, we consider the union of all the objects holistically and mutate them all together. An alternative way is to consider each label one by one. For example, we only mutate all objects belonging to a specific label at one time and then examine whether this label violates the MR. After examining all labels,

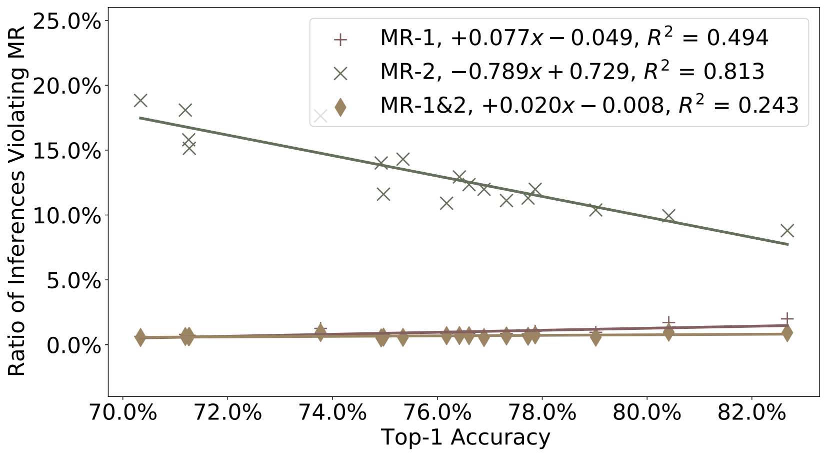


Figure 3.10: The relationship between top-1 accuracy and the ratio of unreliable inferences violating MRs for single-label image classifiers on the ImageNet dataset.

one can conclude whether the inference violates the MR. Such an alternative will increase the workload and requires a more sophisticated methodology to judge whether an inference is reliable based on all its labels. We believe there are several potential ways to define such methodology, thus we leave it as future work to conduct an exhausting study.

Further, for multi-label classification, exact match [190] is used in the comparison of the certainty, *i.e.* $\mathcal{C}(L_{\mathcal{M}(i)}) > \mathcal{C}(L_{\mathcal{M}(i'_c)}) \iff \mathcal{C}_{l,\mathcal{M}(i)} > \mathcal{C}_{l,\mathcal{M}(i'_c)}, \forall\, l \in L_{\mathcal{M}(i)}$. The comparison can use other metrics, such as Hamming Loss and Jaccard Index. In the future, one may investigate the effect of different metrics in the comparison.

3.7.2 Other Potential Application Scenarios

In our study, we focus on the applications of the DNN on image classification. After proper adaption, our MRs can be applied to other applications used on DNN, such as object detection [99, 143, 142] and language processing [41]. For example, in object detection, one may examine the object-relevancy for each detected object. The corresponding MRs can be:[4]

[4]The MR-3/4/5/6 are just our initial proposals. The detailed definition should be polished and their effectiveness should be thoroughly evaluated.

MR-3: An image mutated by corrupting only the features of the target object(s) should lead to an inference result with different label(s) and location(s), or an inference result with the same label(s) and location(s) but with less certainty.

MR-4: An image mutated by preserving the features of the target object(s) and corrupting other features should lead to an inference result with the same label(s) and location(s).

As for language processing, the MR could be:

MR-5: A sentence mutated by corrupting only the content words should lead to a different inference result.

MR-6: A sentence mutated by preserving the content words and corrupting other function words should lead to a similar inference result.

Future work can target proposing new MRs for other DL-based applications and study their effectiveness.

3.7.3 False Positives and False Negatives

The mutations used in our approach can unnecessarily import/remove extra features and then bring some side effects, such as false positives/negatives. Although we applied three mutation operators and adopted the majority voting to alleviate this threat, it still may happen. In the future work, we will explore different image mutation methods and reduce such possible side effects, including false negatives and false positives.

In our evaluation, we only evaluate the effectiveness of our approach from the perspective of true positives and false positives, but not the false negatives, which are the inferences that are based on the object-irrelevant features but are not detected by our approach. It is challenging to identify the false negatives, since it is hard to know whether the inference is indeed completely based on the object-irrelevant features, which is an outstanding challenge in deep learning (see §§3.8.3 and 3.8.4), and whether the changes of the certainty is caused by the imported/removed features in the mutation. We believe it will be one of the future work directions.

3.7.4 The Effect of Annotation Formats

Our metamorphic approach leverages the annotation of the object to construct the follow-up inputs. The availability and the quality of the annotation could affect the performance of our approach. This is the major limitation of our study. As shown in the evaluation in §3.5, inappropriate annotations are the major sources of false positives. In the future work, we will explore new methodologies to alleviate this limitation.

In our study, we use bounding boxes for single-label classification and object masks for multi-label classification, depending on their availability in the datasets. We would like to point out that the annotation format could also affect the effectiveness of our approach. For example, if the annotation is in the format of object mask, even after the object corruption in MR-1, the object shape could still be left in the follow-up inputs, which may cause false positives for MR-1 (similar to the incomplete removal of the target object). According to a recent study [51], the texture of the input image, rather than its shape, has stronger impact in DL-based image classifications. In other words, "*a cat with an elephant texture is an elephant to CNNs, and still a cat to humans*" [51]. Thus, the influence of the shape information left in the follow-up inputs should be limited. Nevertheless, we would like to point out this possible factor and interested researchers may explore along this direction in the future. A possible countermeasure is to develop a novel mutation methodology such that it will further remove the shape information. For example, we can add random padding to the object boundary, so that the image shape information will be destroyed.

3.8 Related Work

3.8.1 Metamorphic Testing in Deep Learning Applications

Several studies have applied metamorphic testing to validate DL applications [195, 42, 45, 208, 172]. Dwarakanath *et al.* [45] leveraged two sets of metamorphic relations to identify faults in machine learning implementations. For example, one metamorphic relation is that the "permutation of input channels (i.e. RGB channels) for the training and test data" would not affect inference results. To validate whether a specific implementation of DL satisfies this relation, they re-ordered the RGB channel of images in both the training set and test set. They examine the impact on the accuracy or precision of the DL model after it is trained using the permuted dataset. Their relations treat the pixels in an image as independent units and they do not consider objects and background in the image.

Xie et al. [195] performed metamorphic testing on two machine learning algorithms: k-Nearest Neighbors and Naïve Bayes Classifier. Their work targets testing attribute-based machine learning models instead of deep learning systems. Ding *et al.* [42] proposed metamorphic relations for DL at three different validation levels: system level, data set level and data item level. For example, a metamorphic relation on system level asserts that DL models should perform better than SVM classifiers for image classification. Their technique requires retraining the systems and is inapplicable to testing pre-trained models.

Other studies [208, 172, 220] leveraged metamorphic testing to validate autonomous driving systems. DeepTest [172] designed a systematic testing approach to detecting the inconsistent behaviors of autonomous driving systems using metamorphic relation. Their relations focus on general image transformation, including scale, shear, rotation and so on. Further, DeepRoad [208] leverages Generative Adversarial Networks to improve the quality of the transformed images. Given an autonomous driving system, DeepRoad mutates the original images to simulate weather conditions such as adding fog to an image. An inconsistency is identified if a DL model and its mutant make an inconsistent decision on an image (e.g., the difference of the steering degrees exceeds a certain threshold). Differently from the existing study, we design metamorphic relations to assess whether an inference is based on object-relevant features for DL-based image classifiers.

3.8.2 Testing Deep Learning Applications

Besides metamorphic testing, studies have also been made to adapt other classical testing techniques for DL applications. A recent survey [207] summarizes the latest work in this direction. DeepXplore [130] proposed neuron coverage to quantify the adequacy of a testing dataset. DeepGauge [104] proposed a collection of testing criteria. TensorFuzz [124], DLFuzz [58] and DeepHunter [196] leveraged fuzz testing to facilitate the debugging process in DL models. DeepMutation [105] applied mutation testing to measure the quality of test data in DNN. Our study falls into the research direction of testing DNN systems. One of our major contributions is that we test DL models from a new perspective, *i.e.* the object relevancy of inferences.

3.8.3 Background Dependence of Computer Vision Systems

Some existing work studied the background dependence of computer vision systems, even before the DL models becomes popular [148, 139]. Qin *et al.* [139] found that removing the background in street scene images can improve the performance of object recognition systems. Rosenfeld *et al.* [149] demonstrated that after transplanting an object from the training set to the background of another image, the state-of-the-art object detectors could fail to identity the inserted object. Later, Wang and Su [183] proposed an automated approach to test the object detectors. Their approach generates test inputs by inserting objects to another image's background. Our study focuses on image classification applications, and we conduct a large-scale empirical study to understand the problem.

3.8.4 Heatmap-based Testing of Deep Learning Applications

Researchers have proposed ideas of generating *HeatMaps* for DL testing and debugging [144, 218, 154, 106, 114, 46]. These HeatMaps essentially capture the *importance* of individual neurons [106] or layers [114, 46] in a given DL model. Based on different definitions of *importance*, these methods generate different types of HeatMaps. Some of them directly use neuron activation values, gradient values *etc.* for HeatMap generation [218, 154]. Others perform some extra processing on such raw data, such as calculating the Jacobian matrix or using differential analysis to extract the differences between correctly classified and misclassified samples [106]. A common drawback of such methods is that there is no standard definition of neuron/layer importance and it is hard to evaluate whether the generated HeatMaps are correct. As a result, these HeatMaps may or may not accurately reflect neuron/layer importance. Compared to their work, the effectiveness of our approach is properly evaluated.

Moreover, some HeatMap generation techniques require the intermediate information from the models and can only be applied for some specific types of models. For example, CAM [218] and GradCAM [218] requires access to the pooling layer of neural networks, which may not always be available. Different from these methods, our method does not need extra intermediate results from models and thus can be applied to any DL-based image classifiers.

3.9 Chapter Conclusion

In this chapter, we proposed to leverage metamorphic testing to identify unreliable image classifications made by DL models based on object-irrelevant features. We proposed two metamorphic relations, from the perspective of object relevancy. We evaluated the effectiveness of our approach and showed that it achieves high precision. We applied our approach to 21 popular pre-trained DL models with the ImageNet and COCO datasets, and found that the phenomenon of unreliable inferences is pervasive. The pervasiveness caused significant bias in model evaluation. Our experiments revealed that the current model training methodologies can guide the models to learn object-relevant features to certain extent, but may not necessarily prevent the model from making unreliable inferences. Therefore, further research is needed to develop a more effective approach for enhancing a model's object-relevancy property.

Chapter 4

Unreliable Deployment of Deep Learning Applications

4.1 Introduction

Compressing DNN models is one *critical* stage in model dissemination, especially for deploying sizable models on mobile or embedded devices with limited computing resources. Compared to their original models, compressed ones achieve similar prediction accuracy while requiring significantly less time, processing power, memory and energy, for inference [34, 182]. However, model compression is a lossy process: given the same input, a compressed model can make predictions deviated from its original model [197, 196]. For example, given the two images in Figure 4.1, the LeNet-4 [88] model correctly predicts both images as `4` while its compressed model predicts the left one as `9` and the right one as `6`. We say that a deviated behavior occurs if a compressed model makes a prediction different from the one of the original model. The input that triggers such a deviated behavior is referred to as a *triggering input*. Our objective is to find the triggering inputs for a given pair of a compressed model and the original one, so that the compressed model's quality can be further assessed before its dissemination beyond the dataset that is used during model compression [4, 98].

It is preferred to find triggering inputs quickly so that developers can obtain in-time feedback to assess and facilitate the entire dissemination workflow. However, this is a challenging task. Specifically, to accelerate the inference speed and reduce storage consumption, compressed models usually do not expose their architectures or intermediate computation results via Application Programming Interfaces (APIs) [34]. Gradient, one of the most

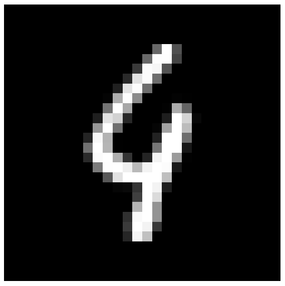
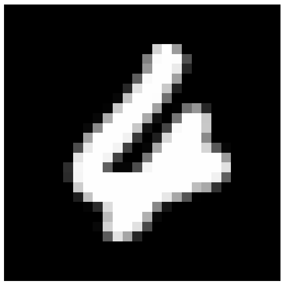



Figure 4.1: Images triggering deviated behaviors between LeNet-4 and its quantized model. The ground truth labels of both images are "4" and both of them are correctly classified as "4" by the original model. However, the quantized model classifies them as "9" and "6" respectively.

common information leveraged by previous test generation approaches [172, 130, 196], is also not always available in compressed models, especially for integer weights (See §4.2.4 for more details). Without such information as guidance, it is difficult for input generation techniques to efficiently find the triggering inputs. For example, the state of the art, DiffChaser [197], requires thousands of queries from a pair of models to find a triggering input. Considering the fact that the datasets in deep learning applications usually consist of more than thousands of inputs, such an inefficient approach not only incurs unaffordable computation workload to developers, but also compromises its practicality in industry.

We propose DFLARE, an effective and efficient technique to automatically find triggering inputs for compressed DNN models that are designed for image classification tasks. Given a non-triggering input as a seed, DFLARE mutates the seed continuously until a triggering input is found. The mutation is guided by a specially designed fitness function, which measures (1) the difference between the prediction outputs of the original and compressed models, and (2) whether the input triggers previously unobserved probability vectors of the two models. The fitness function of DFLARE does not require the model's intermediate results, and thus DFLARE is general and can be applied to any compressed model for image classifications. Unlike DiffChaser, DFLARE only selects one mutation operator and generates one mutated input at each iteration, resulting in much fewer queries than DiffChaser. As another key contribution, DFLARE models the selection of mutation operators as a Markov Chain process and adopts the Metropolis-Hastings (MH) algorithm [75] to guide the selection. Specifically, DFLARE prefers a mutation operator that is likely to increase the fitness function value of subsequent mutated input in the future.

To evaluate DFLARE, we construct a benchmark consisting of 18 pairs of models (*i.e.*, each pair includes the original model and the corresponding compressed one) on three commonly used image classification datasets: MNIST [89], CIFAR-10 [81] and ImageNet [39]. The compressed models are generated with diverse, state-of-the-art techniques: weight pruning, quantization and knowledge distillation. The model architectures include both small- and large-scale ones, from LeNet to VGG-16.

We evaluate DFLARE *w.r.t.* its effectiveness and efficiency and compare it with DiffChaser, the state-of-the-art black-box approach. For effectiveness, we feed a fixed number of seed inputs to DFLARE and measure the ratio of seed inputs for which DFLARE can successfully generate triggering inputs. For efficiency, we measure the time and queries that DFLARE needs to find one triggering input given a seed input. The results show that DFLARE constantly achieves 100% success rate while the baseline DiffChaser fails to do so, whose success rate drops to <90% for certain cases in CIFAR, and drops to around 20% in ImageNet dataset. More importantly, DFLARE can significantly improve efficiency. On average, DFLARE can find a triggering input with only 0.52s and 24.99 queries, while DiffChaser needs more than 52.23s and 3642.50 queries. In other words, the time and queries needed by DFLARE are only 0.99% and 0.69% of DiffChaser, respectively.

We conduct a case study to further demonstrate the usefulness of DFLARE in model dissemination. Specifically, we demonstrate that given a set of compressed models whose accuracy is very close to each other, DFLARE can efficiently provide extra information to approximate the likelihood that the compressed model behaves differently from the original one. Such in-time information can provide developers with more comprehensive evaluations towards compressed models, thus facilitating the selection of compressed models and the compression configurations in the dissemination of image classification models.

We explore the possibility to repair the deviated behaviors using the triggering inputs found by DFLARE. Our intuition is that the substantial amount of triggering inputs found by DFLARE contains essential characteristics of such triggering inputs, and may thus be used to train a separate repair model to fix the deviated behaviors of compressed models found by DFLARE for image classifications. We design a prototype named DREPAIR, serving as a post-processing stage of compressed models. After the compressed model outputs the probability vector of an arbitrary input, DREPAIR takes this vector as input and aims to generate the same label as the one outputted by the original model. We build DREPAIR based on Single-layer Perceptron [150] and train it using the triggering inputs found by DFLARE and seed inputs. Our evaluation shows that DREPAIR can reduce up to 48.48% deviated behaviors and decrease the effectiveness of DFLARE on the repaired models.

Contributions. Our paper makes the following contributions.

1. We propose DFLARE, a novel, search-based, guided testing technique to find triggering

inputs for compressed models for image classifications, to help analyze and evaluate the impact of model compression.

2. Our comprehensive evaluations on a benchmark consisting of 18 pairs of original and compressed image classification models in diverse architectures demonstrate that DFLARE significantly outperforms the state of the art in terms of both effectiveness and efficiency.

3. We demonstrated that the triggering inputs found by DFLARE can be used to repair up to 48.48% deviated behaviors in image classification tasks and decrease the effectiveness of DFLARE on the repaired models.

4. To benefit future research, we have made our source code and benchmark publicly available for reproducibility at `https://github.com/yqtianust/DFlare`

4.2 Preliminary

In this section, we first introduce our scope and give a brief introduction about model compression. Second, we present the annotations and assumptions used in this study and the state-of-the-art technique. At last, we discuss the difference between triggering inputs and adversarial samples.

4.2.1 Scope

Our technique focuses on the compressed DNN models for image classifications. Image classification is one of the most important applications of deep learning and DNN compression techniques. There are enormous studies in model compression focusing on deploying compressed image classification models resource-constrained device, such as [191, 61, 34, 97, 179, 19, 31, 221, 161, 95, 23]. The deployment of compressed models for image classifications is also paid close attention by industries. Mobile hardware vendors, such as Arm [7], Qualcomm[153], and NVIDIA [123] provide detailed documentation to deploy image classification on their mobile devices. Moreover, there are plenty of publicly available original models and compressed models for our research [136, 222] and their detailed instructions allow us to faithfully reproduce their results. Moreover, many previous testing studies for DNN models also primarily focus on image classification tasks [196, 105, 22, 53, 206]. The baseline [197] used in our evaluation also concentrates on the compressed models for image classifications.

Our study aims to find the triggering inputs that are *not* in the original training set or test set. The model compression techniques are designed to compress the original model while preserving the accuracy as much as possible [34]. As a result, the number of triggering inputs in the training set and test set for the original model are pretty limited. If there were a significant number of triggering inputs in the original training set and test set, compressed models are likely to have a clear difference from the original models in terms of accuracy. Developers can easily notice such triggering inputs by inspecting the accuracy and then strive to fix the problematic compression processing before deploying these models. However, the triggering inputs outside the original datasets are not directly available to developers. Finding them can help developers comprehensively evaluate their compressed models before the deployment.

Table 4.1 lists the number of triggering inputs in the training set and test set for three pairs of models used by DiffChaser. The triggering inputs in the training set imply that such deviated behaviors may be related to the inherent proneness of model compression to deviating compressed models from their original models. However, the number of triggering inputs in the training and test set is negligible ($\leq 0.62\%$). These results may mislead the developers of compressed models, *e.g.*, developers may believe that the compressed models have almost identical behaviors as their original models. However, as shown by DiffChaser [197] and later in our evaluation, there are a significant number of triggering inputs that are not in the training set or test set. These extra triggering inputs can help developers comprehensively evaluate their compressed models and repair the deviated behaviors.

Table 4.1: The numbers of triggering inputs and their percentages in training and test set.

Dataset	**Original Model**	**Compression Method**	**Training set**	**Test set**
MNIST	LeNet-1	Quantization-8-bit	83 / 60000 = 0.13%	9 / 10000 = 0.05%
	LeNet-5	Quantization-8-bit	23 / 60000 = 0.38%	5 / 10000 = 0.05%
CIFAR-10	ResNet-20	Quantization-8-bit	75 / 50000 = 0.15%	62 / 10000 = 0.62%

4.2.2 Model Compression

Model compression has become a promising research direction to facilitate the deployment of deep learning models [34]. The objective of model compression is to compress the large model into compact models so that the compressed models are able to be deployed in resource-constrained devices, such as the Internet of Things (IoT) and mobile phones. Various model compression techniques have been proposed to reduce the size of DNN models and the majority of them can be classified into the following three categories.

Pruning. Pruning is an effective compression technique to reduce the number of parameters in DNN models [90, 62]. Researchers find that considerable parameters in DNN models have limited contribution to inference results [34, 90, 62] and removing them does not significantly decrease the model performance on the original test sets. Pruning techniques can be further classified into several categories, according to the subjects to be pruned, including weights, neurons, filters and layers. Weight pruning zeros out the weights of the connections between neurons if the weights are smaller than some predefined threshold. Neuron pruning removes neurons and their incoming and outgoing connections if their contribution to the final inference is negligible. In filter pruning, filters in convolutional layers are ranked by their importance according to their influence on the prediction error. Those least important filters are removed from the DNN models. Similarly, some unimportant layers can also be pruned to reduce the computation complexity of the models.

Quantization. Quantization compresses a DNN model by changing the number of bits to represent weights [217, 141]. In DNN models, weights are usually stored as 32-bit floating-point numbers, After quantizing these weights into 8-bit or 4-bit, the size of models can be significantly reduced. Meanwhile, the quantized models consume less memory bandwidth than the original models. A recent research direction of quantization is Binarization [70, 159]. It uses 1-bit binary values to represent the parameters of DNN models and the model after binarization is referred to as Binarized Neural Networks (BNNs).

Knowledge Distillation. Knowledge distillation transfers the knowledge learned by original DNN models (referred to as teacher models) to compact models (*i.e.*, student models) [18, 133, 111]. After teacher models are properly trained using training sets, student models are trained to mimic the teacher models. We refer interested readers to a recent literature review [54] for details.

4.2.3 Annotations

Let n be the number of all possible classification labels in a single-label image classification problem, *i.e.*, an image is expected to be correctly classified into only one of the n labels. Let f denote a DNN model designed for this single-label image classification, and g denote a corresponding compressed model. Given an arbitrary image as input x, model f outputs a probability vector $f(x) = [p_1, p_2, p_3, \cdots, p_n]$. We refer to the highest probability in $f(x)$

as *top-1 probability* and denote it as $p_{f(x)}$. We refer to the label whose probability is $p_{f(x)}$ in $f(x)$ as *top-1 label* and denote it as $l_{f(x)}$. Similarly, the probability vector of the compressed model, the top-1 probability and its label are denoted as $g(x) = [p'_1, p'_2, p'_3, \cdots, p'_n]$, $p_{g(x)}$ and $l_{g(x)}$, respectively.

4.2.4 Assumptions

We assume that the compressed model g is a black-box and only the information $g(x)$, $p_{g(x)}$ and $l_{g(x)}$ are available [57, 29, 15, 157]. The internal states of models, including intermediate computation results, neural coverages and gradients, are not accessible. We make this assumption for the following reasons.

First, in practice, the intermediate results of compressed models, such as activation values and gradients, are not available due to the lack of appropriate API support in deep learning frameworks. Modern deep learning frameworks, such as TensorFlow Lite [169] and ONNX Inference [125], usually provide APIs only for end-to-end inference of the compressed model, but not for querying intermediate results. The design decision of discarding intermediate results is mainly to improve inference efficiency [34, 125].

Second, gradient information is not generally meaningful for some compressed models. For example, for the model that uses integer weights, their gradients are not defined and thus cannot be acquired. Figure 4.2 shows such an example. The code snippet in Figure 4.2a computes the gradient of $y = x^3$ with respect to the float tensor x and running this code correctly outputs the expected gradient, *i.e.* 12. The code in Figure 4.2b also computes the gradient $y = x^3$ with respect to x, but the tensor x in Figure 4.2b is an integer tensor. Executing the code in Figure 4.2b leads to a runtime error shown in Figure 4.2c.

Third, if the compressed model under test requires special devices such as mobile phones, or the model is compressed on the fly, such as TensorRT [170], accessing the intermediate results requires support from system vendors, which is not always feasible. The assumption of treating compressed models as a black-box increases the generalizability of DFLARE.

4.2.5 State of the Art

DiffChaser [197] is a black-box genetic-based approach to finding triggering inputs for compressed models. In the beginning, it creates a pool of inputs by mutating a given non-triggering input. In each iteration, DiffChaser crossovers two branches of the selected

```
import torch
x = torch.tensor([2.])
x.requires_grad=True
y = x**3
y.backward()
print(x.grad)
```

(a) A code snippet to compute the gradient of `y` *w.r.t.* float tensor `x`.

```
import torch
x = torch.tensor([2.]).int()
x.requires_grad=True
y = x**3
y.backward()
print(x.grad)
```

(b) A code snippet to compute the gradient of `y` *w.r.t.* integer tensor `x`.

```
Traceback (most recent call last):
File "int_gradient.py", line 3, in <module>
x.requires_grad=True
RuntimeError: only Tensors of floating point and complex dtype
can require gradients
```

(c) Error message when executing the code in (b).

Figure 4.2: Example code snippet of computing gradient for tensor with float weight (Figure 4.2a) and integer weight (Figure 4.2b) in PyTorch, respectively. Executing the code in Figure 4.2a outputs the correct value, *i.e.* 12, while executing the code in Figure 4.2b throws runtime error in Figure 4.2c.

inputs and then selectively feeds them back to the pool until any triggering input is found. To determine whether each mutated input will be fed back to the pool, DiffChaser proposes *k-Uncertainty* fitness function and uses it to measure the difference between the highest probability and k-highest probability of *either* $f(x)$ or $g(x)$. Please note that *k-Uncertainty* does not capture the difference between two models, resulting in its ineffectiveness in certain cases, as shown later in §4.5. Another limitation is that the genetic algorithm used in DiffChaser needs to crossover a considerably large ratio of inputs and feed them into DNN models in each iteration. As a result, it requires thousands of queries from the two models to find a triggering input. Such a large amount of queries incur expensive computational resources, which are generally unavailable for devices that have limited computation capabilities, such as mobile phones and Internet-of-Things (IoT) devices.

There are many white-box test generation approaches for a single DNN model [77, 172, 104, 196, 130]. However, they all need to access the intermediate results or gradient to guide their test input generation. Thus, it is impractical to adopt them to address the research problem of this chapter.

4.2.6 Differences from Adversarial Samples

Adversarial samples are different from triggering inputs. The adversarial attack approach targets a *single* model using a malicious input, which is crafted by applying human-imperceptible perturbation on a benign input [22, 53, 124, 130, 205]. In contrast, a triggering input is the one that can cause an inconsistent prediction between *two* models, *i.e.*, the original model and its compressed model. Note that adversarial samples of the original model are often not triggering inputs for compressed models. In our preliminary exploration, we have leveraged FGSM [53] and CW [22] to generate adversarial samples for three compressed models using MNIST. On average, only 18.6 out of 10,000 adversarial samples are triggering inputs. Recent studies pointed out that compressed models can be an effective approach to defend against adversarial samples [28, 76].

4.3 Methodology

This section formulates the targeted problem, and then details how we tackle this problem in DFLARE.

4.3.1 Problem Formulation

Given a non-triggering input as seed input x_s, DFLARE strives to find a new input x_t such that the top-1 label $l_{f(x_t)}$ predicted by the original model f is different from the top-1 label $l_{g(x_t)}$ from the compressed model g, *i.e.*, $l_{f(x_t)} \neq l_{g(x_t)}$. Similar to the mutated-based test generations [197, 124], DFLARE attempts to find x_t by applying a series of input mutation operators on the seed input x_s. Conceptually, $x_t = x_s + \epsilon$, where ϵ is a perturbation made by the applied input mutation operators.

4.3.2 Overview of DFLARE

Algorithm 1: Overview of DFLARE

```
   Input: $x_s$: a seed input
   Input: $f$: the original model
   Input: $g$: the compressed model
   Input: pool: a list of predefined input mutation operators
   Input: $timeout$: the time limit for finding a triggering input
   Output: an triggering input $x_t$
1  $op \leftarrow$ an operator randomly selected from pool
2  $x_{max} \leftarrow x_s$
3  repeat
4      $x \leftarrow op(x_{max})$
5      if $l_{f(x)} \neq l_{g(x)}$ then
6          $x_t \leftarrow x$ // $x_t$ is a triggering input
7          return $x_t$
8      if $H_{f,g}(x) \geq H_{f,g}(x_{max})$ then
9          $x_{max} \leftarrow x$    // if it has higher fitness value
10         $op$.update()    // update its ranking value
11     $op \leftarrow pool$.select($op$)    // select the next operator
12 until $timeout$;
```

Algorithm 1 shows the overview of DFLARE. DFLARE takes four inputs: a seed input x_s, the original model f, the compressed model g, and a list pool of predefined input mutation operators; it returns a triggering input x_t if found.

DFLARE finds x_t via multiple iterations. Throughout all iterations, DFLARE maintains two variables: *op* is the input mutation operator to apply, which is initially randomly picked from `pool` on line 1 and updated each iteration on line 11; x_{max} is the input with the maximum fitness value among all generated inputs, which is initialized with x_s on line 2.

In each iteration, DFLARE applies an input mutation operator on the input which has the highest fitness value to generate a new, mutated input, *i.e.*, $x \leftarrow op(x_{max})$ on line 4. If x triggers a deviated behavior between f and g on line 5, then x is returned as the triggering input x_t. Otherwise, DFLARE compares the fitness values of x and x_{max} on line 8, and use the one that has the higher value for the next iteration (line 9) . The mutation operators are implemented separately from the main logic of DFLARE, and it is easy to integrate more mutation operators. In our implementation, we used the same operators as DiffChaser.

Two factors can significantly affect the performance of Algorithm 1: *fitness function* and *the strategy to select mutation operators*, of which both are detailed in the remainder of this section.

4.3.3 Fitness Function

Following the existing test generation approaches in software testing [27, 124, 197], in DFLARE, if the mutated input x is a non-triggering input, the fitness function $H_{f,g}$ is used to determine whether x should be used in the subsequent iterations of mutation (Algorithm 1, line 8~9). By selecting the proper mutated input in each iteration, we aim to move increasingly close to the triggering input from the initial seed input x_s.

Intuitions of DFLARE

We design the fitness function from two perspectives. First, if x can cause a larger distance between outputs of f and g than x_{max}, x is more favored than x_{max}. The intuition is that if x can, then future inputs generated by mutating x are more likely to further enlarge the difference. Eventually, one input generated in the future will increase the distance substantially such that the labels predicted by f and g become different, and this input is a triggering input that DFLARE has been searching for.

Second, when x and x_{max} cause the same distance between outputs of f and g, x is preferred over x_{max} if x triggers a previously unobserved model state in f or g. Conceptually, a model state refers to the internal status of the original or compressed models during inference, including but not limited to a model's activation status. If an input x triggers a model state that is different from the previously observed ones, it is likely that it triggers a new logic flow in f or g. By selecting such input for next iterations, we are encouraging DFLARE to explore more new logic flows of two models, resulting in new model behaviors, even deviated ones. Since the internal status of compressed models is not easy to collect, we use the probability vector to approximate the model state.

Definition of Fitness Function

Now we present the formal definition of our fitness function $H_{f,g}(x)$ for a non-triggering input x as a combination of two intuitions.

For the first intuition, given an input x, we denote the distance between two DNN models' outputs as $\mathcal{D}_{f,g}(x)$. Since x is a non-triggering input, the top-1 labels of $f(x)$ and $g(x)$ are the same and we simply use the top-1 probability to measure the distance, *i.e.*,

$$\mathcal{D}_{f,g}(x) = |p_{f(x)} - p_{g(x)}| \in [0, 1)$$

For our second intuition, we use the probability vector to approximate the model state. When executing Algorithm 1, we track the probability vectors produced by f and g on all generated inputs. In the calculation of fitness value of x at each iteration, we check whether the pair of probability vectors output by the two DNN models $(f(x), g(x))$ is observed previously or not. Specifically, we adopt the Nearest Neighborhood algorithm [118] to determine $\mathcal{O}(x)$, *i.e.*, whether $(f(x), g(x))$ is close to any previously observed states. The result is denoted as $\mathcal{O}(x)$,

$$\mathcal{O}(x) = \begin{cases} 1 & \text{if } (f(x), g(x)) \text{ has } \textit{not} \text{ been observed} \\ 0 & \text{otherwise} \end{cases}$$

The fitness function $H_{f,g}(x)$ for a non-triggering input x is defined as:

$$H_{f,g}(x) = \delta^{-1} * \mathcal{D}_{f,g}(x) + \mathcal{O}(x)$$

Specifically, according to $H_{f,g}$, for two non-triggering inputs, we choose the one with a higher $\mathcal{D}_{f,g}$ component. If their $\mathcal{D}_{f,g}$ components are very close (*i.e.*, the difference is less than the tolerance δ), they will be chosen based on $\mathcal{O}(x)$. In our implementation, we set $\delta = 1\text{e}{-3}$.

Algorithm 2: Mutation Operator Selection

```
Input: $op_{i-1}$: the mutation operator used in last iteration
Input: pool: a list of predefined input mutation operators
Output: $op_i$: the mutation operator for this iteration
// sort the operators in pool into a list in descending order of the operators'
   ranking values
1 $op\_list \leftarrow$ pool.sort()
2 $k_{i-1} \leftarrow op\_list$.index($op_{i-1}$)
3 $p_{accept} \leftarrow 0$
4 while $random.rand(0,\ 1) \geq p_{accept}$ do
5     $op_i \leftarrow$ a random operator in $op\_list$
6     $k_i \leftarrow op\_list$.index($op_i$)
7     $p_{accept} \leftarrow (1-p)^{k_i - k_{i-1}}$
8 return $op_i$
```

4.3.4 Selection Strategy of Mutation Operators

Existing work on test generation for traditional software has shown that the selection strategy of mutation operators can have a significant impact on the performance of mutation-based test generation techniques adopted by DFLARE [86, 27]. Following prior work, in each iteration, DFLARE favors a mutation operator that has a high probability to make the next mutated input x have a higher fitness value than x_{max}. Unfortunately, it is non-trivial to obtain such prior probabilities of mutation operators before the mutation starts.

To tackle the challenge of selecting effective mutation operators, DFLARE models the problem as a Markov Chain [110] and uses Monte Carlo [75] to guide the selection. During the test generation, DFLARE selects one mutation operator from a pool of operators and applies it to the input. This process can be modeled as a stochastic process $\{op_0, op_1, \cdots, op_t\}$, where op_i is the selected operator at i-th iteration. Since the selection of op_{i+1} from all possible states only depends on op_i [86, 186, 27], this process is a typical Markov Chain. Given this modeling, DFLARE further uses Markov Chain Monte Carlo (MCMC) [75] to guide the selection of mutation operators in order to mimic the selection from the actual probability distribution.

Specifically, DFLARE adopts Metropolis-Hasting algorithm [75], a popular MCMC method to guide the selection of mutation operators from the operator pool. Throughout all iterations, for operator *op*, DFLARE associates it with a ranking value:

$$v(op) = \frac{N_i}{N_{op} + \epsilon}$$

where N_{op} is the number of times that operator *op* is selected and N_i is the number of times that the fitness value of input is increased after applying *op*. $\epsilon = 1e - 7$ is used to avoid division by zero when $N_{op} = 0$. These numbers are dynamically updated in the generation as shown in Algorithm 1, line 10.

The detailed algorithm for the operator selection given the operator at last iteration op_{i-1} in DFLARE is shown in Algorithm 2. Based on each operator's ranking value v, DFLARE first sorts the mutation operators in the descending order of v (line 1) and denotes the index of op_{i-1} as k_{i-1} (line 2). Then DFLARE selects one mutation operator from the pool (line 5) and calculates the acceptance probability for op_i given op_{i-1} (line 7):

$$P\left(op_i | op_{i-1}\right) = (1 - p)^{k_i - k_{i-1}}$$

where p is the multiplicative inverse for the number of mutation operators in the pool. Following the Metropolis-Hasting algorithm, DFLARE randomly accepts or rejects this mutation operator based on its acceptance probability (line 7). The above process will repeat until one operator is accepted.

4.4 Experiment Design

In this section, we introduce the design of our evaluation. In particular, we aim to answer the following four research questions in our evaluation.

RQ1 Is DFLARE effective to find triggering inputs?

RQ2 Is DFLARE time-efficient and query-efficient to find triggering inputs?

RQ3 What are the effects of the fitness function and the selection strategy of mutation operator used by DFLARE in finding triggering inputs?

RQ4 Can DFLARE facilitate the dissemination of compressed models?

RQ5 Can the triggering input found by DFLARE be used to repair the deviated behaviors?

We collect 18 pairs of original models and their compress models to answer the effectiveness and efficiency of DFLARE in **RQ1** and **RQ2**. For **RQ3**, we conduct an ablation study to understand the impacts of our fitness function and mutation operation selection strategy on effectiveness and efficiency. In **RQ4**, we design a case study and discuss one potential application of DFLARE to facilitate model dissemination. For **RQ5**, we explore the possibility to repair the deviated behaviors of compressed models using the triggering input found by DFLARE.

4.4.1 Datasets and Seed Inputs

We use the three datasets: MNIST [89], CIFAR-10 [81] and ImageNet [39] to evaluate the performance of DFLARE. We choose them as they are widely used for image classification tasks, and there are many models trained on them so that we can collect a sufficient number of compressed models for evaluation. These datasets are also used by many studies in model compression [191, 61, 34, 97, 179, 19, 31, 221, 161, 95]. For each dataset, we randomly select 500 images as seed inputs from their test set for evaluation. Each seed input in MNIST and CIFAR-10 is pre-processed by normalization based on the mean value *mean* and standard deviation *std* of the dataset, *i.e.*, $\frac{x_s - mean}{std}$. For the inputs in ImageNet, they are pre-processed using the function provided by each model. To mitigate the impact of randomness, we repeat the experiments five times and each time use a different random seed.

4.4.2 Compressed Models

The compressed models used in our evaluation come from two sources. First, we use three pairs of the original model and the according quantized model used by DiffChaser: LeNet-1 and LeNet-5 for MNIST, and ResNet-20 for CIFAR-10. They are compressed by the authors of DiffChaser using TensorFlow Lite [169] with 8-bit quantization. The upper half of Table 4.2 shows their top-1 accuracy.

Second, to comprehensively evaluate the performance of DFLARE on other kinds of compressed techniques, we also prepare 15 pairs of models. Specifically, six of them are for MNIST and nine of them are for CIFAR-10. These compressed models are prepared by three kinds of techniques, namely, quantization, pruning, and knowledge distillation, using Distiller, an open-source model compression toolkit built by the Intel AI Lab [222]. The remaining three models for ImageNet and their quantized models are collected from PyTorch [136]. These three models are chosen since their accuracy is highest among all compressed models in PyTorch Models. The lower half of Table 4.2 shows their top-1 accuracy.

4.4.3 Evaluation Metrics

For effectiveness, we measure the success rate to find a triggering input for selected seed inputs. In terms of efficiency, we measure the average time and model queries it takes to find a triggering input for each seed input. All of them are commonly used by related studies [197, 57, 130, 124]. Their details are explained as follows.

Success Rate. It measures the ratio of the seed inputs based on which a triggering input is successfully found over the total number of seed inputs. The higher the success rate, the more effective the underlying methodology. Specifically,

$$\text{Success Rate} = \frac{\sum_{i=1}^{N} s_{x_i}}{N}$$

where s_{x_i} is an indicator: it is equal to 1 if a triggering input based on seed input x_i is found. Otherwise, s_{x_i} is 0. N is the total number of seed inputs, *i.e.*, 500 in our experiments.

Average Time. It is the average time to find a triggering input for each seed input. Mathematically,

$$\text{Average Time} = \frac{\sum_{i=1}^{N} t_{x_i}}{N}$$

where t_{x_i} is the time spent to find a triggering input given the seed input x_i. The shorter the time, the more efficient the input generation. We measure the average time spent to find all triggering inputs provided the seed inputs.

Average Query. It measures the average number of model queries issued by DFLARE to find a triggering input for each seed input. Formally, this metric is defined as:

$$\text{Average Queries} = \frac{\sum_{i=1}^{N} q_{x_i}}{N}$$

where q_{x_i} is the number of queries to find a triggering input given the seed input x_i. A model query means that one input is fed into both the original DNN model and the compressed one. Since the computation of the DNN models is expensive, it is preferred to issue as few queries as possible. The fewer the average queries, the more efficient the test generation.

Table 4.2: The top-1 accuracy of the original models and compressed models used in the evaluation. The first three models are from DiffChaser and the other models are prepared by this study.

Dataset	**Original Model**	**Accuracy**(%)	**Compression Method**	**Accuracy**(%)
MNIST	LeNet-1	97.88	Quantization-8-bit	97.88
	LeNet-5	98.81	Quantization-8-bit	98.81
CIFAR-10	ResNet-20	91.20	Quantization-8-bit	91.20
MNIST	CNN	99.11	Pruning	99.23
			Quantization	99.13
	LeNet-4	99.21	Pruning	99.13
			Quantization	99.21
	LeNet-5	99.13	Pruning	98.99
			Quantization	99.15
CIFAR-10	PlainNet-20	87.33	Knowledge Distillation	75.89
			Pruning	85.98
			Quantization	87.12
	ResNet-20	89.42	Knowledge Distillation	74.60
			Pruning	89.88
			Quantization	88.89
	VGG-16	87.48	Knowledge Distillation	87.59
			Pruning	88.44
			Quantization	87.06
ImageNet	Inception	93.45	Quantization	93.35
	ResNet-50	95.43	Quantization	94.98
	ResNeXt-101	96.45	Quantization	96.33

4.4.4 Experiments Setting

Baseline and its Parameters. We use the DiffChaser [197] as the baseline, since it is the state-of-the-art black-box approach to our best knowledge. Specifically, we use the source code and its default settings provided by the corresponding authors. For the timeout to find triggering inputs for each seed input, we use the same setting as DiffChaser, *i.e.* 180s. The experiment platform is a CentOS server with a CPU 2xE5-2683V4 2.1GHz and a GPU 2080Ti.

Mutation Operators. For a fairness evaluation, we used the same image mutation operators from the baseline DiffChaser, as shown in Table 4.3. These mutation operators are proposed by prior work [130, 197, 172, 104, 196] to simulate the scenario that DNN models are likely to face in the real world. For example, Gaussian Noise is considered as one of the most frequently occurring noises in image processing [17]. After applying each mutation operator to a given image, we clip the values of pixels to $[0, 255]$ so that the resulted images are still valid images. Please note that these mutation operators may have certain randomness. For example, the size of the average filter used by *Average Blur Image* is randomly selected from 1 to 5.

4.5 Evaluation Results and Analysis

4.5.1 RQ1: Effectiveness

Triggering Inputs found by Dflare

Figure 4.3 shows three examples of the triggering inputs found by Dflare in MNIST, CIFAR-10, and ImageNet respectively. The original models correctly classify the two

Table 4.3: Mutation operators used in Dflare and DiffChaser

Category	Mutation Operator	Description
Adding Noise	Random Pixel Change	Randomly change the values of pixels to arbitrary values in $[0, 255]$
	Gaussian Noise	Generate a random Gaussian-distributed noise [52] and add it into the image.
	Multiplicative Noise	Generate a random Multiplicative noise [52] and add it into the image
Blurring Image	Average Blur Image	Blur the image using a random average filter.
	Gaussian Blur Image	Blur the image using a random Gaussian filter.
	Median Blur Image	Blur the image using a random median filter.

inputs as “5”, “cat”, “great white shark” respectively. However, the inputs are misclassified as “6”, “deer” and “marimba” (a musical instrument) by the associated compressed models, respectively.

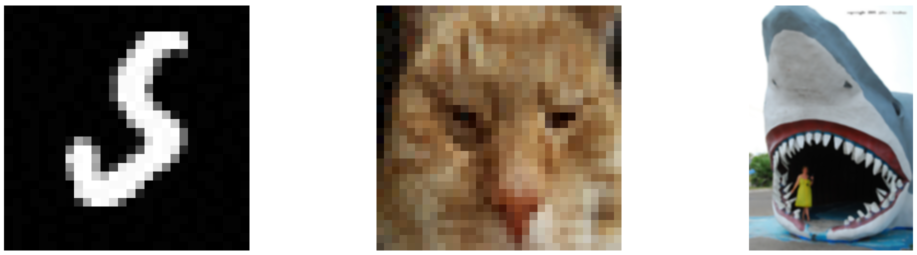


Figure 4.3: Triggering inputs found by DFLARE.

Success Rate

The two **Average Success Rate** columns in Table 4.4 show the success rate of DFLARE and DiffChaser, respectively. DFLARE achieves 100% success rate for all pairs of models on three datasets. As for DiffChaser, its success rate on MNIST and CIFAR-10 datasets, ranges from 74.12% to 99.92%, with an average of 96.39%. Such results indicate that DiffChaser fails to find the triggering input for certain seed inputs of all the pairs. Specifically, the success rate of DiffChaser is lower than 90% for three Quantization Model in the CIFAR-10 dataset, while DFLARE constantly achieves 100% success rate in all models. For the models that are trained on ImageNet, the success rates of DiffChaser range from 12.01% to 21.12%. This result demonstrates that DFLARE outperforms DiffChaser in terms of effectiveness. The reason is that DiffChaser, especially its *k-Uncertainty* fitness function, does not properly measure the differences between two models, resulting in failures to find triggering input for certain cases. In contrast, the fitness function of DFLARE not only measures the differences between the prediction outputs of the original and compressed models, but also measures whether the input triggers previously unobserved states of two models. By combining this fitness function with our advanced selection strategy of mutation operators, our approach always achieves 100% success rates in our experiments.

To further investigate the effectiveness of DFLARE, we feed all the non-triggering inputs in the entire test set as seed inputs into DFLARE on the 18 pairs of models. We found that DFLARE can consistently achieve a 100% success rate for all 18 pairs. The result on ImageNet models is in shown in Table 4.5 and it shows that DFLARE is effective to find the triggering inputs for these large models trained on complex dataset and the success rates

Table 4.4: Comparison of effectiveness and time-/query-efficiency between DFLARE and DiffChaser. The results are averaged across five runs using different random seeds.

Dataset	Model	Compression	DFLARE Average Success Rate	DFLARE Average Time (sec)	DFLARE Average Query	DiffChaser Average Success Rate	DiffChaser Average Time (sec)	DiffChaser Average Query
MNIST	LeNet-1	Quantization-8-bit	100%	0.513	83.97	99.40%	10.654	5812.47
	LeNet-5	Quantization-8-bit	100%	0.706	117.02	99.68%	12.598	6040.53
CIFAR-10	ResNet-20	Quantization-8-bit	100%	0.509	30.43	99.76%	33.980	2323.58
MNIST	LeNet-4	Prune	100%	0.056	18.34	99.44%	16.249	6172.57
		Quantization	100%	0.187	27.83	98.08%	76.254	6506.53
	LeNet-5	Prune	100%	0.071	22.03	98.56%	17.446	6276.38
		Quantization	100%	0.225	28.08	98.48%	45.618	6662.88
	CNN	Prune	100%	0.068	22.51	99.60%	16.381	6053.82
		Quantization	100%	0.173	25.34	99.52%	38.039	6450.96
CIFAR-10	PlainNet-20	Prune	100%	0.051	4.31	99.80%	18.222	1896.59
		Quantization	100%	0.470	9.13	89.52%	75.191	1696.16
		Knowledge Distillation	100%	0.029	3.97	99.72%	12.324	1961.09
	ResNet-20	Prune	100%	0.063	4.70	99.88%	23.298	2145.77
		Quantization	100%	0.685	10.16	74.12%	83.971	1511.06
		Knowledge Distillation	100%	0.032	3.91	99.92%	14.117	2097.28
	VGG-16	Prune	100%	0.041	5.84	99.60%	15.619	2453.01
		Quantization	100%	1.183	26.16	80.08%	85.709	2129.37
		Knowledge Distillation	100%	0.036	5.78	99.80%	16.058	2761.12
ImageNet	Inception	Quantization	100%	1.266	21.44	20.20%	163.847	1808.87
	ResNet-50	Quantization	100%	0.819	19.24	21.12%	158.393	1702.10
	ResNeXt-101	Quantization	100%	3.693	34.49	12.01%	163.936	2030.41

Table 4.5: Effectiveness of DFLARE on on ImageNet models using entire test set as seed inputs. The inputs in the ImageNet test set that can trigger deviated behaviors are excluded from experiments. The results are averaged across five runs.

Dataset	Model	Accuracy	Compression	Accuracy	**Average Success Rate**	**Average Time (sec)**	**Average Query**
ImageNet	Inception	93.45%	Quantization	93.35%	100%	1.22	18.54
	ResNet-50	95.43%	Quantization	94.98%	100%	0.80	15.19
	ResNeXt-101	96.45%	Quantization	96.33%	100%	4.33	28.22
		Average			100%	2.12	20.65

are 100% in five runs. Due to the low efficiency of DiffChaser as shown in the next section, we are not able to conduct the same experiments using DiffChaser.

Answer to RQ1: DFLARE is effective in finding triggering inputs for compressed models. Specifically, it constantly achieves 100% success rate in all 18 pairs of models.

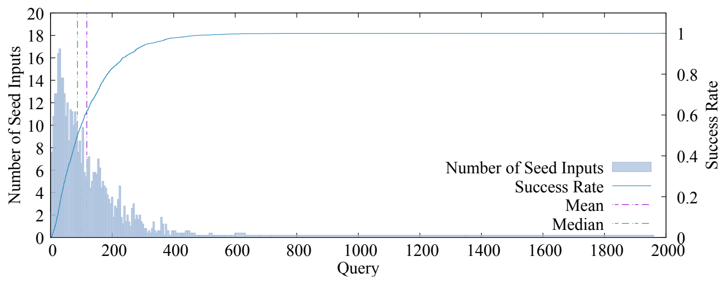
4.5.2 RQ2: Efficiency

Time

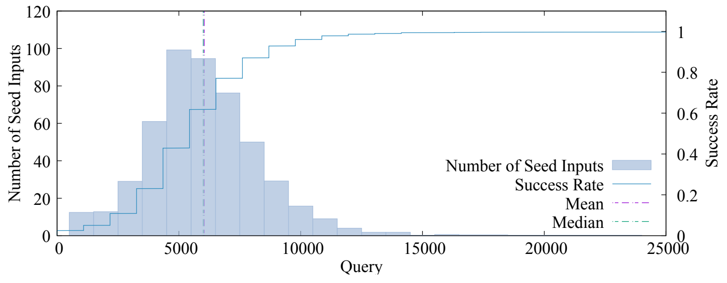
The two **Average Time** columns in Table 4.4 show the average time spent by DFLARE and DiffChaser to find triggering inputs for each seed input if successful. The time needed by DFLARE to find one triggering input ranges from 0.029s to 3.369s, with the average value 0.518s. DiffChaser takes much longer time than DFLARE. Specifically, DiffChaser takes 10.654s∼163.936s to find one triggering input, with the average 52.234s. On average, DFLARE is 230.94x (17.84x∼446.06x) as fast as DiffChaser in terms of time.

Query

The two **Average Query** columns in Table 4.4 show the average query issued by DFLARE and DiffChaser for all seed inputs if a triggering input can be found. Generally, DFLARE only needs less than 30 queries to find a triggering input, with only two exceptions. On average, DFLARE requires only 24.99 queries (3.9∼117.0). DiffChaser always needs thousands of queries for each trigger input (averagely 3642.50), much more than DFLARE. For example,



(a) DFLARE

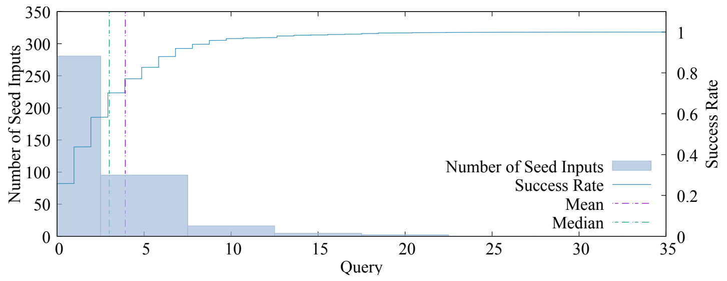


(b) DiffChaser

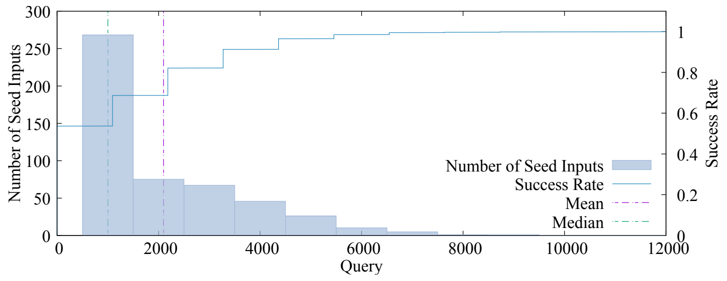
Figure 4.4: Histogram of the number of queries required by DFLARE to find the triggering input for the given seed input on LeNet5 Quantization-8-bit models. The value is averaged over five repeated experiments.

the smallest number of queries needed by DiffChaser is 1,896.59 for PlainNet-20 and its pruned model. In the same pair of models, DFLARE only needs 4.31 queries on average. Overall, the number of queries required by DFLARE is 0.699% (0.186%∼1.937%) of the one required by DiffChaser.

We further visualize the queries of DFLARE and DiffChaser in Figures 4.4 and 4.5 on two pairs of models: LeNet-5 Quantization-8-bit and ResNet-20 Knowledge Distillation. They are selected since the ratio of queries needed by DFLARE over the one needed by DiffChaser is the smallest (0.186%) and largest (1.937%) in all the 18 pairs of models. Each figure shows the histogram of the number of queries needed by DFLARE or DiffChaser, as well as the mean and median. It can be observed that DFLARE significantly outperforms DiffChaser in terms of queries. The reason is that DiffChaser adopts a genetic algorithm to generate many inputs via crossover and feed them into DNN models in each iteration. As a result, it requires thousands of queries from the two models to find a triggering input. In contrast, DFLARE only needs to generate one mutated input and query once in each iteration.



(a) DFLARE



(b) DiffChaser

Figure 4.5: Histogram of the number of queries required by DiffChaser to find the triggering input for the given seed input on ResNet-20 Knowledge Distillation models. The value is averaged over five repeated experiments.

Answer to RQ2: DFLARE is efficient to find triggering inputs in terms of both time and queries. On average, DFLARE is 230.94x as fast as DiffChaser and takes only 0.699% queries as DiffChaser.

4.5.3 RQ3: Ablation Study

We further investigate the effects of our fitness function and mutation operator selection strategy. Specifically, we create the following two variants of DFLARE.

1. DFLARE$_D$: the fitness function in DFLARE is replaced by a simpler fitness function: $H_{f,g}(x) = \mathcal{D}_{f,g}(x) = |p_{f(x)} - p_{g(x)}|$. In other words, the fitness function does not trace the model states triggered by inputs.
2. DFLARE$_R$: the selection strategy for mutation operators in DFLARE is changed to uniform random selection.

For each variant, we measure its success rate, computation time, and the number of queries needed using the seed inputs of the preceding experiments. Table 4.6 shows the results. The numbers in parentheses are the ratios of time or queries spent by each variant with respect the one(s) spent by DFLARE.

Fitness Function

The column **DFLARE$_D$** in Table 4.6 shows the evaluation results of DFLARE$_D$. Although DFLARE$_D$ still achieves 100% success rate in half of the 18 model pairs, the success rates of DFLARE$_D$ for the remaining 21 pairs are clearly lower than those of DFLARE, ranging from 39.44% to 99.04%. The average success rate of DFLARE$_D$ over all 18 pairs of models is only 83.14%. In terms of the computation time and the number of queries, DFLARE$_D$ is much less efficient than DFLARE. Specifically, the time spent by DFLARE$_D$ is 1.140x∼168.58x of that spent by DFLARE, with an average value 50.06x. As for the number of queries needed, the ratios range from 1.18x to 193.20x, and the average ratio is 54.85x. This result indicates the importance of encouraging the mutated inputs to explore more model states as formulated by our fitness function.

Table 4.6: Evaluation results of DFLARE$_D$ and DFLARE$_R$. The numbers in parentheses are the ratios of time or queries spent by each variant with respect the one spent by DFLARE. The results are averaged across five runs using different random seeds.

Dataset	Model	Compression	DFLARE$_D$			DFLARE$_R$		
			Average Success Rate	**Average Time (sec)**	**Average Query**	**Average Success Rate**	**Average Time (sec)**	**Average Query**
MNIST	LeNet-1	Quantization-8bit	41.44%	39.988 (77.93)	8140.54 (96.94)	100%	0.537 (1.05)	89.52 (1.07)
	LeNet-5	Quantization-8bit	39.44%	38.244 (54.15)	7422.38 (63.43)	100%	0.752 (1.07)	125.87 (1.08)
CIFAR-10	ResNet-20	Quantization-8bit	100%	3.454 (6.78)	203.91 (6.70)	100%	0.555 (1.09)	30.59 (1.01)
MNIST	LeNet-4	Prune	92.12%	8.242 (148.23)	2970.80 (161.98)	100%	0.062 (1.12)	20.79 (1.13)
		Quantization	60.76%	31.162 (166.55)	4861.92 (174.69)	100%	0.200 (1.07)	30.06 (1.08)
	LeNet-5	Prune	93.52%	7.428 (104.62)	2473.71 (112.27)	100%	0.080 (1.13)	24.77 (1.12)
		Quantization	59.92%	26.699 (118.72)	3548.56 (126.38)	100%	0.237 (1.05)	29.99 (1.07)
	CNN	Prune	81.04%	11.396 (168.58)	4348.22 (193.20)	100%	0.075 (1.11)	25.14 (1.12)
		Quantization	63.68%	26.639 (154.16)	4243.05 (167.46)	100%	0.182 (1.05)	26.39 (1.04)
CIFAR-10	PlainNet-20	Prune	100%	0.037 (1.28)	5.16 (1.30)	100%	0.031 (1.05)	4.05 (1.02)
		Quantization	100%	0.058 (1.14)	5.11 (1.18)	100%	0.051 (1.00)	4.32 (1.00)
		Knowledge Distillation	100%	0.809 (1.72)	15.85 (1.74)	100%	0.477 (1.02)	9.30 (1.02)
	ResNet-20	Prune	100%	0.039 (1.22)	4.95 (1.27)	100%	0.031 (0.98)	3.80 (0.97)
		Quantization	100%	0.080 (1.26)	6.07 (1.29)	100%	0.066 (1.03)	4.89 (1.04)
		Knowledge Distillation	100%	1.329 (1.94)	19.39 (1.91)	100%	0.690 (1.01)	10.07 (0.99)
	VGG-16	Prune	100%	0.047 (1.28)	7.86 (1.36)	100%	0.035 (0.95)	5.49 (0.95)
		Quantization	100%	0.050 (1.20)	7.48 (1.28)	100%	0.042 (1.01)	5.90 (1.01)
		Knowledge Distillation	99.04%	4.986 (4.22)	117.08 (4.48)	100%	1.133 (0.96)	24.75 (0.95)
ImageNet	Inception	Quantization	79.4%	16.064 (12.69)	251.55(11.73)	100%	1.459 (1.15)	25.01 (1.17)
	ResNet-50	Quantization	87.4%	12.789 (15.63)	253.57(13.18)	100%	1.418 (1.73)	20.34 (1.06)
	ResNeXt-101	Quantization	48.2%	29.019 (7.86)	277.17(8.04)	100%	4.240 (1.15)	40.64 (1.18)
	Average Ratio *w.r.t.* DFLARE			50.06	54.85		1.09	1.05

Selection Strategy of Mutation Operator

The column **DFLARE**$_R$ in Table 4.6 shows the evaluation results of DFLARE$_R$. Same as DFLARE, DFLARE$_R$ achieves 100% success rate. In terms of efficiency, the average time spent by DFLARE$_R$ is 1.09x (0.95x∼1.73x) of that spent by DFLARE. The ratio of queries required by DFLARE$_R$ over those by DFLARE is also 1.05x, ranging from 0.95x to 1.18x. In 17 out of 18 pairs, the time and queries required by DFLARE are 91.33% of that required by DFLARE$_R$. For the remaining 4 pairs, DFLARE$_R$ is marginally (3.5%) more efficient than DFLARE in terms of time and the number of queries. A possible reason is that with our fitness function, a triggering input for these four pairs can be found in just a few iterations. In such cases, the selection strategy of DFLARE has not obtained enough samples to capture the knowledge of each mutation operator before the triggering input is found. Therefore, it is possible that DFLARE$_R$, which adopts a random mutation strategy with our effective fitness function, can find the triggering inputs sooner.

To check whether DFLARE statistically outperforms DFLARE$_R$ in terms of time, we conduct Wilcoxon significant test [189] and the p-value is 3.604×10^{-4}. The p-value indicates that our MH algorithm for mutation operator selection significantly improves the efficiency of finding triggering inputs.

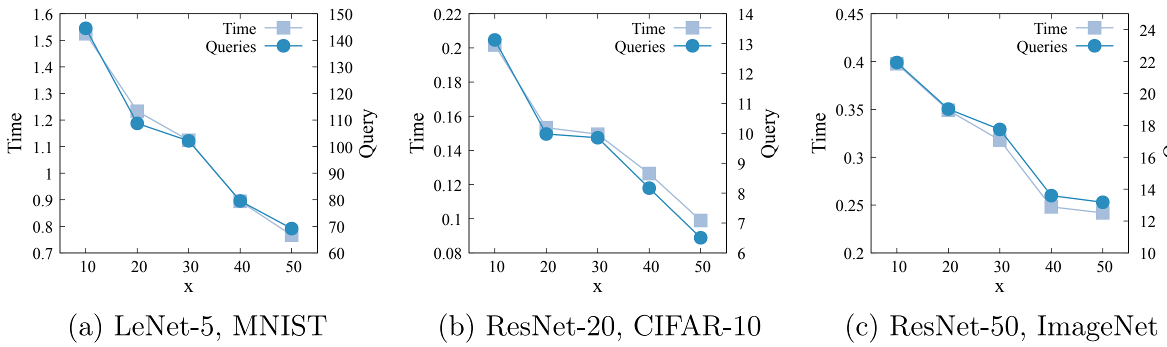
Answer to RQ3: Our fitness function and selection strategy both contribute to the effectiveness and efficiency of DFLARE.

4.5.4 Application of DFLARE: Facilitating Model Dissemination

In this case study, we discuss a potential application of DFLARE to facilitate model dissemination. Specifically, we are going to show that, to a certain extent, the time and number of queries taken to find triggering inputs can be leveraged as an approximation of to what extent the behavior of compressed models differs from that of the original models in the dissemination. Since DFLARE can provide this metric effectively and efficiently, we argue that DFLARE is able to provide developers with in-time feedback complementary to the accuracy metric, to assess compressed models.

Correlation

We would like to understand the correlation between *the time and queries* and *to what extent the behavior of compressed models differs from that of the original models in de-*



(a) LeNet-5, MNIST (b) ResNet-20, CIFAR-10 (c) ResNet-50, ImageNet

Figure 4.6: Correlation between time/query and mutation ratio x, using LeNet-5 for MNIST (Figure 4.6a), ResNet-20 for CIFAR-10 (Figure 4.6b), and ResNet-50 for ImageNet (Figure 4.6c).

ployment. We manually constructed a series of models from the original models LeNet-5, ResNet-20 and ResNet-50 in Table 4.2 by mutating x% of weights, where x ranges from 10 to 50, with a step of 10. In the mutation, we randomly mutated the x% of the weights by increasing or decreasing their values by 10%. Intuitively, the larger x is, the more likely the behavior of the resulted model differs from the one of original model. These models serve as a benchmark with the ground truth, *i.e.*, to what extent the resulted model differs from the original one, for our study. Then we applied DFLARE using the same experiment settings in §4.4.4 and measured the time and number of queries.

Figures 4.6a to 4.6c show the results for LeNet-5, ResNet-20, and ResNet-50, respectively. Success rates are not presented since all of them are 100%. It is clear that as the portion of the mutated weight x% increases, the time and queries required to find the triggering inputs decrease. The Pearson Correlation Coefficients [14] between x and time/-queries also confirm this strong negative correlation, which are -0.989 (time) and -0.972 (queries) for LeNet-5, -0.968 and -0.967 for ResNet-20, and -0.979 and -0.977 for ResNet-50, respectively. Since the higher x causes the resulted model to be more likely to differ from the original models, we claim that the time and queries can approximate to what extent the behavior of compressed models differs from the one of original models. Specifically, the less time and fewer queries needed to find triggering inputs, the more likely the compressed model differs from the original model in the dissemination.

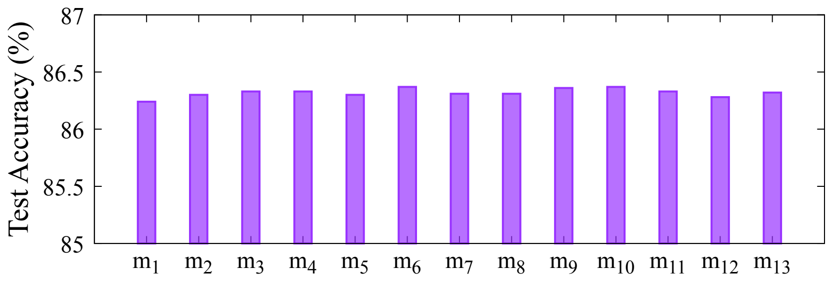


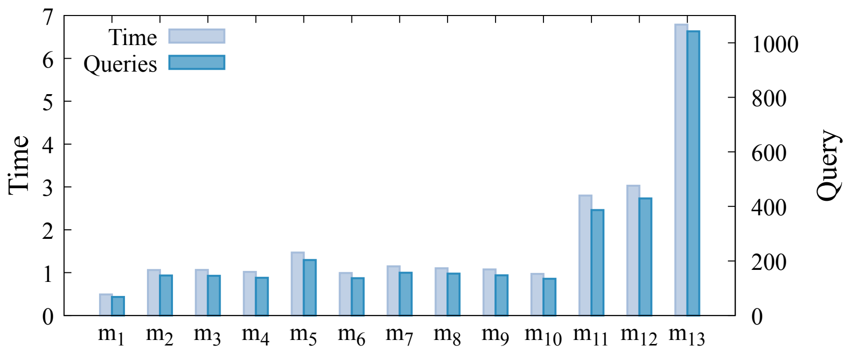
Figure 4.7: The test accuracy of thirteen compressed models.

Application

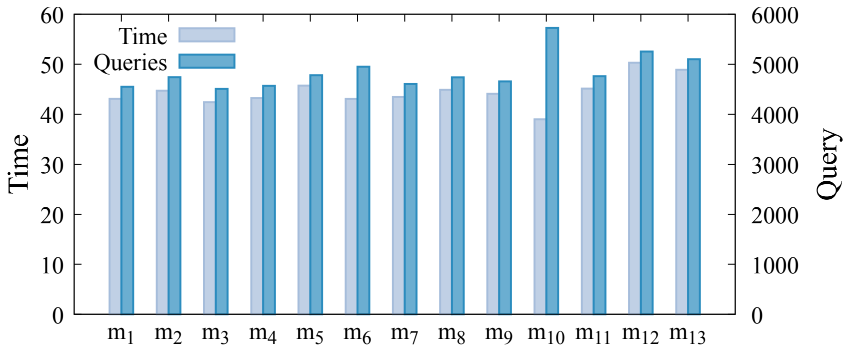
Now we present an application of DFLARE in model dissemination. When compressing a pre-trained model, developers often need to prepare a compression configuration [90, 34]. For example, the configuration of model pruning usually specifies which layers in the original model are to be pruned. A common way is to select the configuration that produces the highest accuracy on test set. However, as we will demonstrate, only using accuracy is insufficient to distinguish different models, and DFLARE can provide complementary information to facilitate this process.

We prepared a VGG-16 model by training it from scratch using the CIFAR-10 training dataset. After the loss and accuracy became saturated, its top-1 accuracy on the CIFAR-10 test set is 86.34%. Given this original model, we created a set of compressed models by pruning only one of the thirteen convolutional layers in the VGG-16 model at one time. In total, we collected thirteen compressed models using PyTorch and we referred to them as m_1, m_2, $\cdots$, m_{13}, where m_i is the compressed model obtained by pruning the i-th convolutional layer of the original model. Figure 4.7 shows the top-1 accuracy of each compressed model. The accuracy of these models ranges from 86.24% to 86.37% and is almost identical to the accuracy of the original model (86.34%) with a maximal difference of 0.10%. If this developer uses accuracy as the single evaluation metric, it seems that these models achieve indistinguishable performance, and thus it makes no difference to select any of them for dissemination.

In this scenario, DFLARE can quickly provide complementary information that is orthogonal to accuracy. Figure 4.8a shows the average time and queries when using DFLARE to find one bug-triggering input. Same as the previous settings, we repeated each experiment five times using 500 seed inputs. Although the accuracy of these models is similar, the information generated by DFLARE leads to a different conclusion. Specifically, it is relatively harder to find a deviated behavior for the compressed model whose pruned layer



(a) DFLARE



(b) DiffChaser

Figure 4.8: The results using DFLARE and DiffChaser on thirteen compressed models.

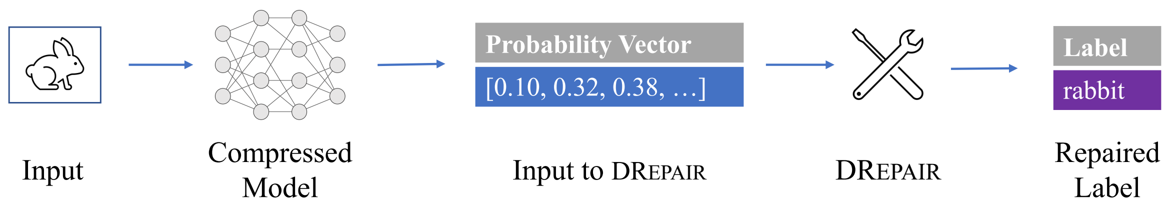
is at the bottom of VGG-16, than the models whose pruned layer is at the top. For example, m_{13} requires much more time and queries than m_1. According to the aforementioned correlation, if we use the time and number of queries as an approximation of the likelihood that the compressed model behaves differently from the original model, it is clear that m_{13} has the least likelihood among all thirteen models. Taking account of the perspectives from both accuracy and this likelihood information provided by DFLARE, the developers should choose the compressed model m_{13} or m_{12} for dissemination, since they have not only the comparable accuracy, but also the least likelihood to exhibit deviated behaviors.

Figure 4.8b shows the results generated by DiffChaser. The average success rate of DiffChaser is only 86.3%, which is 13.7% lower than DFLARE. The time and number of queries required by DiffChaser demonstrate the same trend as the one using DFLARE, *i.e.*, the models whose pruned layers are at the bottom of the VGG16, *e.g.* m_{13}/m_{12}, are less likely to have deviated behaviors than others, *e.g.* m_1/m_2. DFLARE can provide such in-time feedback to developers due to its high effectiveness and efficiency, making it practical to utilize this technique in daily tasks. In contrast, even though DiffChaser may also provide similar information, it takes much a longer time (37.4x on average) and more queries (29.74x) to do so, imposing large computation cost. For example, given a set of 500 seed inputs and m_2, DiffChaser requires 6.1 hours and 2,370,800 queries, while DFLARE only needs 8.9 minutes and 73,320 queries.

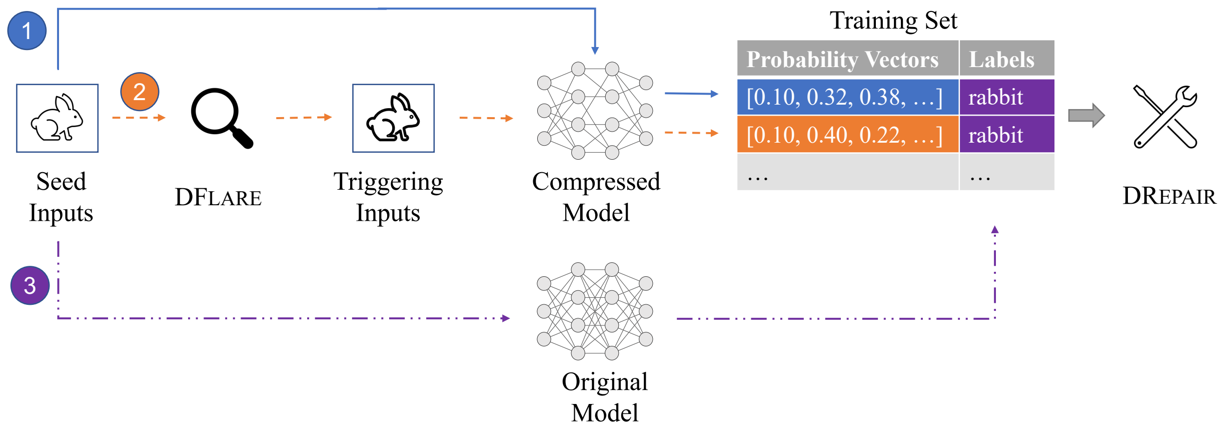
Answer to RQ4: DFLARE can provide developers with in-time feedback complementary to the accuracy metric, to assess compressed models.

4.5.5 Application of DFLARE: Repairing the Deviated Behaviors

We further explored the possibility to repair the deviated behaviors of the compressed models for image classification models using the triggering inputs found by DFLARE. A common approach to improving the performance of DNN models is to retrain the DNN models. For example, adversarial training can improve the robustness of DNN models [206, 155]. However, without accessing the internal architectures and status of compressed models, it is difficult to repair the deviated behaviors directly via retraining. Therefore, we explored an alternative approach that repairs the deviated behaviors without the need to retrain the compressed model. Please note that we are not attempting to repair the triggering inputs in the original test sets, since the number of triggering inputs in the original test set is ineligible, as shown in Algorithm 2. It is the duty of compression techniques to reduce the number of triggering inputs in the original test set, to avoid accuracy degradation due to



(a) Deployment Stage



(b) Training Stage

Figure 4.9: The workflow of DREPAIR to repair deviated behaviors of compressed model.

model compression.

Design of DREPAIR

We proposed a prototype, DREPAIR, to repair the deviated behaviors of the compressed models for image classifications. Our intuition is that the substantial amount of triggering inputs found by DFLARE contains essential characteristics of such triggering inputs, and may thus be used to train a separate repair model to fix the deviated behaviors. Figure 4.9a illustrates the workflow of DREPAIR. DREPAIR is a supervised classifier, serving as a post-processing stage of the target compressed model. Given an input x and the probability vector $g(x) = [p_1, p_2, p_3, \cdots, p_n]$ outputted by a compressed model g, DREPAIR takes as input the probability vector $g(x)$ and is expected to output a label $\overline{l_{g(x)}}$ such that $\overline{l_{g(x)}} = l_{f(x)}$, where the label $l_{f(x)}$ is outputted by the original model f given the same input x.

Figure 4.9b shows the workflow to train DREPAIR. After collecting a set of seed inputs, we first ① feed each seed input x_s to the compressed model under test g and collect the probability vector $g(x_s)$. Then we ② utilize DFLARE to find the triggering input x_t given the seed input x_s and obtain its probability vector $g(x_t)$ using the compressed model g. Since DREPAIR is a supervised classifier, each vector in the training set is assigned a target label. For vector $g(x)$, we ③ use the label outputted by the original model f given input x as the target label, *i.e.* $l_{f(x)}$. This is because the objective of DREPAIR is to produce a label that is the same as the label from original model.

Implementation and Evaluation of DREPAIR

We implemented DREPAIR using a Single-layer Perceptron (SLP), *i.e.*, a neural network with a single hidden layer [150]. We chose SLP since it is light-weight in terms of computational resources and thus is applicable to be deployed along with compressed models in embedded systems. We used five-fold cross-validation to evaluate the performance of DREPAIR using the seed inputs and triggering inputs found by DFLARE in RQ2. Specifically, for each set of 500 pairs of seed input and triggering input, we collected their probability vectors and split them into five portions of equal size. We chose four of them for the training set of DREPAIR, and the remaining one as its test set. In other words, each training set contains 400 non-triggering inputs and 400 triggering inputs, and each test set X contains 100 non-triggering inputs and 100 triggering inputs. In a five-fold cross-validation, we repeated the training and testing five times and ensured a different training and test set is used in each time. Each five-fold cross-validation was conducted 5 times using different random seeds.

We measure the performance of DREPAIR from the following three perspectives. In particular, we use X_t to denote the set of triggering input and use X_s to denote non-triggering inputs in the test set X. As we mentioned in the above paragraph, the sizes of X_t, X_s and X are 100, 100 and 200, respectively.

Repair Count and ***Repair Ratio***. We first use *Repair Count* to measure the number of triggering inputs in X_t that do not trigger deviated behaviors *after* repair, *i.e.*,

$$\text{Repair Count} = \sum_{i=1}^{|X_t|} o_{x_i}$$

where o_{x_i} is an indicator and $o_{x_i} = 1$ only if $\overline{l_{g(x_i)}} = l_{f(x_i)}$, *i.e.*, x_i does not trigger deviated behavior after repair; otherwise, it is 0. $|X_t|$ is the number of inputs in X_t.

We then measure *Repair Ratio*, *i.e.*, the ratio of Repair Count in X_t. *Repair Ratio* measures the percentage of triggering inputs in X_t that do not trigger deviated behaviors after repair. The higher the repair ratio is, the more triggering inputs are repaired by DREPAIR.

$$\text{Repair Ratio} = \frac{\text{Repair Count}}{|X_t|} \times 100\% = \frac{\sum_{i=1}^{|X_t|} o_{x_i}}{|X_t|} \times 100\%$$

Inducing Count and ***Inducing Ratio***. We use *Inducing Count* to denote the number of non-triggering inputs in X_s that trigger deviated behaviors *after* repair, *i.e.*

$$\text{Inducing Count} = \sum_{i=1}^{|X_s|} k_{x_i}$$

where k_{x_i} is an indicator and $k_{x_i} = 1$ only if $\overline{l_{g(x_i)}} \neq l_{f(x_i)}$, *i.e.*, x_i triggers deviated behaviors after repair; otherwise, it is 0. $|X_s|$ is the number of inputs in X_s.

Then we use *Inducing Ratio* to measure the ratio of Inducing Count in X_s. Specifically, *Inducing Ratio* measure the percentage of non-triggering inputs in X_s that trigger deviated behaviors after repair. The lower the inducing ratio is, the fewer deviated behaviors are induced by DREPAIR.

$$\text{Inducing Ratio} = \frac{\text{Inducing Count}}{|X_s|} = \frac{\sum_{i=1}^{|X_s|} k_{x_i}}{|X_s|} \times 100\%$$

Improvement Count and ***Improvement Ratio***. We use *Improvement Count* to measure the difference between the number of deviated behaviors in X *before* repair by DREPAIR and the number of deviated behaviors in X *after* repair. Specifically, the number of deviated behaviors *before* the repair is equal to the number of triggering inputs in X, *i.e.* $|X_t|$. A deviated behavior *after* repair is triggered by x_i if $\overline{l_{g(x_i)}} \neq l_{f(x_i)}$. Since the indicator $k_{x_i} = 1$ if and only if $\overline{l_{g(x_i)}} \neq l_{f(x_i)}$, the number of deviated behaviors *after* repair in X is counted as $\sum_{i=1}^{|X|} k_{x_i}$. Therefore, the difference between the number of deviated behaviors in X before repair and after repair is denoted as

$$\text{Improvement Count} = |X_t| - \sum_{i=1}^{|X|} k_{x_i}$$

We then use *Improvement Ratio* to measure the ratio of Improvement Count *w.r.t.* to the number of deviated behaviors in X *before* repair. The higher the improvement ratio is, the more effective DREPAIR is to repair the deviated behaviors of compressed models.

$$\text{Improvement Ratio} = \frac{\text{Improvement Count}}{|X_t|} \times 100\% = \frac{|X_t| - \sum_{i=1}^{|X|} k_{x_i}}{|X_t|} \times 100\%$$

Noticed that the Improvement Ratio can be zero or negative when the number of triggering inputs after repair is equal to or larger than the number of triggering inputs before repair, *i.e.*, Improvement Count ≤ 0. In such situations, the repair process fails since the number of deviated behavior after repair is more than or equal to the number of deviated behavior before repair.

Table 4.7 shows the results. On average, DREPAIR repairs 32.73% triggering inputs. Although DREPAIR induces 4.17% new deviated behaviors, overall DREPAIR reduces the number of deviated behaviors by 30.16%. In the best case, the number of deviated behaviors is reduced by 48.48%. In conclusion, it is feasible to repair the deviated behaviors using the triggering inputs found by DFLARE. A promising feature work is to propose more advanced approaches to achieve this objective.

We further leveraged DFLARE to test these models that are repaired by DREPAIR. Specifically, we selected the three models that have the highest improvement ratios to see if these models that are relatively successfully repaired by DREPAIR can decrease the effectiveness or efficiency DFLARE. Meanwhile, we also selected the three models trained on ImageNet to investigate the effects of DREPAIR in large and complex models. Table 4.8 shows the effectiveness and efficiency of DFLARE when the compressed model is not repaired by DREPAIR and when the compressed model is repaired by DREPAIR. After repair, DFLARE can still achieve 100% success rates in these six models. However, the time spent by DFLARE to find each triggering input in the compressed model repaired by DREPAIR is 2.27x∼5.87x as the one spent by DFLARE on the compressed models without repair. The number of queries is also increased to 1.46x∼4.54x as the one without repair. As a proof of concept proposed by us, DREPAIR can effectively decrease the efficiency of DFLARE. We will make the efforts to improve the effectiveness of DREPAIR as our following work.

Answer to RQ5: DREPAIR reduces the deviated behaviors up to 48.48% and decreases the efficiency of DFLARE. This result demonstrates the feasibility to repair the deviated behaviors using the triggering inputs found by DFLARE. We call for contributions from the community to propose more advanced approaches.

Table 4.7: Evaluation results of DREPAIR. The results are averaged across five-fold cross-validation. Noticed that since X_s and X_t are equal to 100 in each fold of validation, the Count = Ratio $\times$ 100.

Dataset	**Model**	**Compression**	DREPAIR Average Repair Count	DREPAIR Average Repair Ratio	DREPAIR Average Inducing Count	DREPAIR Average Inducing Ratio	DREPAIR Average Improvement Count	DREPAIR Average Improvement Ratio
MNIST	LeNet-1	Quantization-8-bit	30.56	30.56%	0.36	0.36%	30.20	30.20%
	LeNet-5	Quantization-8-bit	24.28	24.28%	0.12	0.12%	24.16	24.16%
CIFAR-10	ResNet-20	Quantization-8-bit	15.04	15.04%	0.52	0.52%	14.52	14.52%
MNIST	LeNet-4	Prune	48.48	48.48%	0.00	0.00%	48.48	48.48%
		Quantization	35.68	35.68%	0.00	0.00%	35.68	35.68%
	LeNet-5	Prune	43.20	43.20%	0.08	0.08%	43.12	43.12%
		Quantization	38.68	38.68%	0.00	0.00%	38.68	38.68%
	CNN	Prune	42.44	42.44%	0.04	0.04%	42.40	42.40%
		Quantization	36.55	36.55%	0.04	0.04%	36.50	36.50%
CIFAR-10	PlainNet-20	Prune	50.40	50.40%	12.92	12.92%	37.48	37.48%
		Quantization	32.20	32.20%	8.16	8.16%	24.04	24.04%
		Knowledge Distillation	23.34	23.34%	6.77	6.77%	16.56	16.56%
	ResNet-20	Prune	49.32	49.32%	12.24	12.24%	37.08	37.08%
		Quantization	36.28	36.28%	6.44	6.44%	29.84	29.84%
		Knowledge Distillation	24.78	24.78%	6.38	6.38%	18.40	18.40%
	VGG-16	Prune	37.68	37.68%	5.12	5.12%	32.56	32.56%
		Quantization	25.64	25.64%	6.88	6.88%	18.76	18.76%
		Knowledge Distillation	19.28	19.28%	4.78	4.78%	14.50	14.50%
ImageNet	Inception	Quantization	28.69	28.69%	7.26	7.26%	21.43	21.43%
	ResNet-50	Quantization	18.94	18.94%	4.01	4.01%	14.93	14.93%
	ResNeXt-101	Quantization	25.92	25.92%	5.44	5.44%	20.48	20.48%
Average			32.73	32.73%	4.17	4.17%	28.56	28.56%

Table 4.8: The effectiveness and efficiency of DFLARE on the compressed model without DREPAIR and with DREPAIR. The numbers in parentheses are the ratios of time or queries spent by DFLARE on the compressed model repaired by DREPAIR with respect the one spent by DFLARE on the model without DREPAIR. The results are averaged across five runs using different random seeds.

Dataset	Model	Compression	Improvement Ratio	Without DREPAIR			With DREPAIR		
				Average Success Rate	**Average Time (sec)**	**Average Query**	**Average Success Rate**	**Average Time (sec)**	**Average Query**
MNIST	LeNet-4	Prune	48.48%	100%	0.056	18.34	100%	0.245 (4.83)	55.67 (3.04)
	LeNet-5	Prune	43.12%	100%	0.071	22.03	100%	0.169 (2.38)	33.86 (1.54)
	CNN	Prune	42.40%	100%	0.068	22.51	100%	0.399 (5.87)	102.29 (4.54)
ImageNet	Inception	Quantization	21.44%	100%	1.266	21.44	100%	2.880 (2.27)	31.40 (1.46)
	ResNet-50	Quantization	18.94%	100%	0.819	19.24	100%	2.959 (3.61)	39.45 (2.05)
	ResNeXt-101	Quantization	25.92%	100%	3.693	34.49	100%	9.872 (2.67)	55.06 (1.60)

4.6 Discussion and Future Work

4.6.1 Demonstration of the Generalizability of DFLARE on Other Domain

Our study focuses on the compressed models for image classifications, but our approach can also be applied to the compressed models in other domains after proper adaptions, especially the mutation operators. To demonstrate this, we applied DFLARE to the compressed models on Speech-to-Text task. Given an audio clip as input, Speech-to-Text models aim to translate the audio into text. We used the original models and compressed models provided by Mozilla DeepSpeech [63, 117]. We selected Mozilla DeepSpeech since it is a well-recognized open-source project (with more than 20,000 stars) and it provides detailed documentation for us to deploy. There are two pairs of original models and compressed model used in our evaluation. Specifically, the latest version of DeepSpeech, *i.e.*, v0.9.3, provides a pair of original model and compressed models and the second latest version, v0.8.2, provides the second pair of models (versions between these two versions provide the same models as v0.9.3). In both version, the compressed models are quantized from the original models.

We adjusted DFLARE in two aspects to apply it in Speech-to-Text models. First, we adopted the audio-specific mutation operators since audio and images have different characteristics. Specifically, we used the operators TimeStretch, PitchShift, TimeShift, and Gain (volume adjustment) provided by Audiomentations [9], a Python library to mutate

Table 4.9: Effectiveness of DFLARE on on Speech-to-Text models. The results are averaged across five runs.

Model	Version	Compression	DFLARE Average Success Rate	DFLARE Average Time (sec)	DFLARE Average Query	DiffChaser Average Success Rate	DiffChaser Average Time (sec)	DiffChaser Average Query
DeepSpeech	v0.9.3	Quantization	100%	5.740	8.42	95.6%	169.365	223.04
	v0.8.2	Quantization	100%	4.812	5.88	95.5%	168.402	214.53

audio. Since these operators are also used in DeepSpeech for data augmentation during model training [116], we believe that these operators are regarded as representative mutations by developers. Second, since the output of Speech-to-Text models is a sentence, rather than a label in image classifications, we also adjusted the methodology to compare the outputs of original models and compressed models. Specifically, in image classification models, DFLARE compares the labels outputted by original models and compressed models, while in Speech-to-Text, DFLARE compares the sentences word by word. Given the same audio, if the original model and compressed model output different sentences, such as "the character which your royal highness *assumed* is imperfect harmony with your own" vs "the character which your royal highness *summed* is imperfect harmony with your own", such an audio input is labeled as triggering input. We also made the same adjustment to the baseline DiffChaser. Three authors carefully reviewed the adjustment to avoid possible mistakes.

We randomly selected 500 audio inputs from the test set of Librispeech dataset [126]. According to the documentation of DeepSpeech, Librispeech is used by Mozilla DeepSpeech in training and testing. We used the same timeout as RQ1, *i.e.*, 180 seconds. The experiments were repeated five times using different random seeds.

Table 4.9 shows results. The success rates of DFLARE are 100% in five runs. On average, it takes DFLARE 4.812∼5.740s and 5.88∼8.42 queries to find a triggering input. By contrast, DiffChaser fails to find triggering inputs for around 4.5% seed inputs and it takes DiffChaser around 168 seconds and 214.53∼223.04 queries to find one triggering input. The time and queries spent by DFLARE is only 2.4%∼3.4% and 2.7%∼3.8% of the one required by DiffChaser, respectively. This result demonstrates the effectiveness and efficiency of DFLARE on Speech-to-Text tasks.

We also tried to fix the triggering inputs using DREPAIR but we were not able to achieve a reasonable result. Our conjecture is that repairing the models trained for Speech-to-Task are much more complicated than the models trained for image classifications. Specifically, for image classification models trained on ImageNet, DREPAIR is expected to output the label that is same as the label outputted by the original model from 1,000 candidate labels (since ImageNet has 1,000 image labels). In contrast, for the Speech-to-Text task, the output of models is a sentence that can have an arbitrary number of words and there are around 977,000 unique English words in Librispeech. To successfully repair the results outputted by compressed models, DREPAIR needs to not only select a correct set of words from these 977,000 words, but also make sure these words are in the proper order since the meaning of a sentence also depends on the order of words. As a simple prototype proposed by us, DREPAIR is not able to handle such a complicated scenario. We leave the improvement of DREPAIR of large datasets like Librispeech for future work.

4.6.2 Effect of Timeout

In our evaluation, we used 180s as timeout for both DFLARE and DiffChaser. To understand the effect of timeout on the effectiveness of DFLARE, we conducted further experiments using smaller timeouts. Specifically, we evaluated the success rate of DFLARE using 15s, 10s and 5s. Our experiments covered all the pairs of models in RQ2 and used all the images from MNIST and CIFAR-10 test sets as seed inputs.

DFLARE achieves 100% success rates for the 14 pairs of models out of 18 pairs even using 5s as the timeout. Table 4.10 shows the results of the remaining four pairs of models. The success rates of DFLARE for these four pairs drop to different levels when the timeout is shortened. The most significant decrease comes from the PlainNet-20 and its quantized model. Specifically, its success rate drops to 76.93% when the timeout is set to 15s. The success rate drops further to 10.90% with 5s timeout. The success rate for VGG-16 and its compressed model also drops to 40.06% with 5s timeout. As for the other two pairs of models in Table 4.10, their success rates slightly decrease to 99.98% and 89.12% if 5s timeout is used, respectively. In summary, DFLARE is effective for 16 out of 18 pairs of models even a short timeout such as 10s is used.

An interesting observation from Table 4.10 is that all the compressed models in Table 4.10 are compressed using 8-bit quantization. A possible explanation is that the difference between an original model and its compressed model induced by quantization is relatively smaller than the difference induced by pruning and knowledge distillation. Therefore, it takes a relatively long time for DFLARE to find the deviated behavior for quantized models.

Table 4.10: The success rates of DFLARE using different timeouts. The pairs of models that have 100% success rate using 5s timeout are not included. The results are averaged across five runs.

Dataset	**Model**	**Compression**	Timeout		
			15s	10s	5s
CIFAR-10	ResNet-20	Quantization-8-bit	100%	100%	99.98%
CIFAR-10	PlainNet-20	Quantization	76.93%	43.76%	10.90%
	ResNet-20	Quantization	100%	99.30%	89.12%
	VGG-16	Quantization	95.87%	79.30%	40.06%

4.6.3 Uniqueness of Triggering Inputs

We carefully checked the triggering inputs found by DFLARE in Table 4.4. Specifically, we first represented each triggering input x as a matrix A_x with size $H \times W \times C$, where H and W are the height and width of the image, respectively, and C refers to the number of channels of the image ($C = 3$ in color images and $C = 1$ in gray images). Please note that the pixels in images are integers in the range $[0, 255]$, and thus the elements in A_x are also integer numbers in the range $[0, 255]$. For each triggering input x, we check if there exists a triggering input y such that the matrix A_x is equal to the matrix A_y. If such y exists, the inputs x and y are labeled as duplicated triggering inputs. Otherwise, x is a unique triggering input.

Out of 105 experiments (18 pairs of models $\times$ 5 runs), 77 experiments do not have any duplicated triggering inputs. For the remaining 28 experiments, on average, 99.04% of the triggering inputs are unique to each other. In other words, the vast majority of the triggering inputs found by DFLARE are unique.

4.6.4 Future Work

A useful future work is to fix the deviated behaviors for compressed DNN models. As we showed in §4.5.5, there is still a significant improvement space for the performance of our repair prototype. A promising research direction is to propose an effective and efficient approach for this issue.

In §4.6.1, we demonstrate the generalizability of DFLARE in Speech-to-Text tasks. A promising direction is to apply DFLARE to other domains, such as natural language process [41] and object detection [143]. To achieve this, the mutation operators should be properly customized based on domain-specific knowledge. Meanwhile, the test oracle may be adjusted accordingly, since the DNN models in other domains may concern factors other than labels. For example, in object detection, the location and boundary of the detected object are also important [66]. Moreover, a sufficient number of compressed models and datasets from the AI community are critical to comprehensively evaluate the new techniques in other domains. We believe it is a fruitful working direction to explore.

Another potential follow-up direction is to leverage DFLARE to directly test the DNN models deployed on the embedded or mobile platforms. This may help the developers reveal the deviated behaviors induced by the hardware or firmware of such platforms.

DFLARE may also be improved in various ways. For example, DFLARE adopts the top-1 probability to measure the distance between the outputs of two models. Future work may explore other distance metrics, such as L_p distance [22] (including L_0, L_1 and L_∞) and K-L divergence [73]. The efficiency of DFLARE could be further improved by proposing new mutation operators and designing effective selection strategies for mutation operators.

4.7 Threats to Validity

4.7.1 Internal Threats

First, both DFLARE and DiffChaser have randomness at certain levels. Such randomness may affect the evaluation results. To alleviate this, all experiments are repeated five times using different random seeds and the average results are presented. We found that the variance across these five runs are low and the conclusions of our evaluation are consistent in each run, *i.e.*, DFLARE outperforms DiffChaser in both effectiveness and efficiency. Therefore, we did not run the experiments more times.

Second, to comprehensively evaluate DFLARE using diverse compressed DNN models, we construct the first benchmark containing 18 pairs of original model and its compressed model. Since there are no published pairs of the original model and its compressed model for 15 out of 18 pairs, we prepare them based on popular compression algorithms. Specifically, we train the DNN models from scratch and then compress them using popular model compression techniques. Both processes may be affected by the randomness in deep learning at a certain level [132]. To mitigate this threat, for model training, we follow the practice from AI community and train each model until the loss value is saturated. We also compare their accuracy with the one reported by their original publications. The accuracy of each model trained by us is close to its published accuracy. In order to make sure that the model compressed by us are valid evaluation subjects, we utilize an existing tool from Intel AI Lab [222] and carefully follow the instructions. The accuracy of each compressed model is close to that of its original model. This suggests that our compression processes are reliable.

Third, it is possible that the triggering inputs found by DFLARE do not comply with the real world data distribution. To alleviate such a problem, DFLARE used the mutation operators from prior work [130, 197, 172, 104, 196] and these mutation operators are designed to simulate the scenario that DNN models are likely to face in the real world. For example, the mutation operator *Random Pixel Change* simulates effects of "dirt on camera lens" [130]. Gaussian Noise is one of the most frequently occurring noises in image signal [17]. Therefore, the triggering inputs found by DFLARE using these mutation operators are highly likely to comply with the real-world inputs to be fed to DNN models in model deployment.

Lastly, since the implementation of DiffChaser shared by its authors only supports image classification models, we carefully revised its source code to support Speech-to-Text models in §4.6.1. It is possible that our revision might have mistakes and thus affects its effectiveness and efficiency. To address this threat, three authors carefully reviewed the changes made by us to avoid possible mistakes.

4.7.2 External Threats

We evaluate our approach using 18 compressed models. The selection may not cover all compression techniques proposed by the communities. To mitigate this, the models selected are representative as they are trained on two common datasets at different scales, and then compressed using popular model compression techniques. Besides, the architectures of the selected models are diverse and include the ones that are commonly used by existing studies [130, 172, 171].

4.8 Related Work

4.8.1 Testing Deep Learning Applications

DeepXplore [130] is the first technique targeted at testing DNN models. It proposed neuron coverage, which measures the activation state of neurons, to guide the generation of test inputs. DeepXplore is based on differential testing and it uses multiple models of a task to detect potential defects. To alleviate the need of multiple models under test, DeepTest [172] leverages metamorphic relations [24] that are expected to hold by a model as its test oracles. Both DeepXplore and DeepTest perturb their test inputs based on the gradient of deep learning models. TensorFuzz [124] and DeepHunter [196] are whitebox fuzzing-based testing techniques. They guide the input mutation by certain predefined coverage, instead of gradient, in order to trigger the unexpected behaviors of deep learning models, *e.g.* numerical errors and classifications. To assess the quality of DNN models, DeepJanus [146] proposes the notion of *frontier of behaviors*, *i.e.*, pairs of inputs that have different predictions from the same DNN model. Given a DNN model under test, DeepJanus leverages a multi-objective evolutionary approach to find the frontier of behaviors. It further utilizes the model-based input representation to assure the realism of generated inputs.

Our approach, DFLARE, differs from these techniques in two ways. First, DFLARE focuses on the deviated behaviors of compressed models, while existing techniques target the normal models. Second, the majority of existing testing techniques for DNN models are whitebox [130, 172, 24, 124, 196], making use of the models' internal states, such as gradients and neuron coverage, which are often unavailable for compressed models. Therefore, these techniques are not applicable to testing compressed DNN models. In contrast, our approach is specifically designed for compressed models and it does not require the internal information from the model under test. The black-box testing approaches, *e.g.*, DeepJanus, with proper adaptations, are promising to be applied in finding deviated behaviors of compressed DNN models. We will explore this direction in the future work.

Besides DiffChaser, there are also several recent studies specifically targeting on compressed models. DiverGet [201] presents a search-based approach to assess quantization models for hyperspectral images. It proposes a set of domain-specific metamorphic relations to transform the hyperspectral images and use them to mutate hyperspectral images. BET [181] is a testing method for convolutional neural network(CNN)s. It splits a convolutional kernel into multiple zones of which the weights have the same positive or negative signs. The insight is that the decisions of CNNs are likely to be affected by continuous perturbations, *i.e.*, the perturbations that have the same sign with each zone. These two approaches are either specific to the compression methods (quantization model in DiverGet) or types of DNN (CNN in BET), while DFLARE is a general approach for diverse types of model architectures and compression methods. We do not include their approach in our evaluation since their tools are not available.

4.8.2 Empirical Study on the Deployment Issues of Deep Learning Applications

Researchers have conducted several empirical studies to characterize the issues in deploying DNN models, including compressed DNN models. Guo *et al.* found that the DNN models deployed in other platforms may exhibit different behaviors from the original models [59]. Hu *et al.* conducted a deep analysis for quantization models [68]. They found that retraining the compressed models with triggering inputs cannot effectively reduce the behavioral difference between the original model and the compressed one. Our approach, DFLARE, is a testing technique for compressed models, with the aim to help developers address these issues in model deployment and dissemination. Using the triggering inputs found by DFLARE, our prototype DREPAIR is able to repair up to 48.48% deviated behaviors.

4.8.3 Differential Testing

DFLARE aims to find deviated behavior between two DNN models. Related works also include those applying differential testing to detect inconsistencies across two pieces of traditional software. McKeeman [109] originally proposed differential testing in 1998 to expose bugs in software systems using test cases that result in inconsistent execution results in multiple comparable systems. Le *et al.* [85] introduced EMI, which applies differential testing on compilers using semantically equivalent programs. Inconsistent execution outputs of compiled programs may indicate defects in compilers. Further, differential testing is also applied in JVM implementations [96] using mutated Java bytecode [26].

The objective of DFLARE is similar to differential testing. Rather than two pieces of code, the systems under test for DFLARE are DNN models and their compressed ones.

4.8.4 Differential Verification of Deep Learning Applications

ReluDiff [128] and its following work [129] share certain similar objectives with our approach although it is not a testing technique. It leverages the structural and behavioral similarities of the two closely related networks in parallel, to verify whether the output difference of the two models are within the specification. In the evaluation, they use the pairs of compressed model and the original model as subjects.

Our work differs from ReluDiff in two ways. First, ReluDiff can only be used in forward neural networks with relu activation function for both compressed and original models. This limits its application scenarios. Sophisticated DNN models usually contain convolutional layers and recurrent layers. The advantage of DFLARE is that it makes no assumption on the model architecture, making it applicable to a wide range of application scenarios. Second, ReluDiff needs to know the architectures and weights of DNN models for verification, while DFLARE works for black-box models.

4.9 Chapter Conclusion

In this chapter, we proposed DFLARE, a novel, effective input generation method to find deviated behaviors between an original DNN model and its compressed model. Specifically, DFLARE leverages the MH algorithm in the selection of a mutation operator at each iteration to successively mutate a given seed input. DFLARE incorporates a novel fitness function to determine whether to use a mutated input in subsequent iterations. The results show that DFLARE outperforms prior work in terms of both effectiveness and efficiency. DFLARE constantly achieves 100% success rate but uses significantly less amount of time and queries than the state of the art. We also explored the possibility to repair such deviated behaviors using the triggering inputs found by DFLARE. Our prototype DREPAIR can repair up to 48.48% deviated behaviors and decrease the effectiveness of DFLARE on the repaired models.

Chapter 5

Unreliable Assessment of Deep Learning Applications

5.1 Introduction

Compilers are among the most important, fundamental system software. Every program, no matter whether it is an operating system or application, has to be compiled by a compiler from source code into binary executable, so that the program can be executed by a computer. Hence the reliability of compilers is critical, especially in the era of digitalization. To this end, automated compiler testing is an active research area which aims to automatically find bugs in production compilers. Various language-specific methodologies [202, 102] have been proposed to generate random programs to test whether a compiler can correctly compile these programs; if the compiler crashes or hangs (*i.e.*, the compilation process does not terminate normally), or the compiled binary behaves differently from the source code, then a compiler bug is found. For example, Csmith [202] is designed to randomly generate well-defined C programs[1] and can generate only C programs. Csmith has helped find hundreds of bugs in GCC and LLVM, and therefore has been integrated into the daily testing routine of GCC. However, it is non-trivial, time-consuming, and labor-intensive to design and implement such a *language-specific* program generator (referred to as LSG), which requires comprehensive language-specific domain knowledge to design correct, subtle program generation rules.

[1]A program is well-defined if the program does not contain any undefined behaviors; and an undefined behavior is a behavior that is not defined in the C language standard [71].

To minimize the cost of engineering a random program generator for compiler testing, researchers recently resorted to deep learning techniques [100, 37]. Such a deep learning-based program generator (referred to as DLG) attempts to automatically learn the syntactic and semantic rules of a programming language from human-written programs using a sophisticated deep learning model. Later, the DLG uses the trained model to generate programs to test the compiler of this language. Creating a DLG does not involve human efforts to encode domain knowledge into program generation rules, and this approach is demonstrated to be effective in compiler testing compared to LSGs. For example, DeepFuzz [100] claimed that it triggers more code paths in GCC than Csmith.

However, we argue that it is unfair to use LSGs as the baseline to evaluate DLGs. An LSG usually has complex generation rules to ensure the generated programs are compilable and well-defined, in order to validate whether compilers correctly optimize and produce binary code. These generation rules inevitably restrict the diversity of language features used in the generated test programs. For example, Csmith-generated programs only cover a small subset of the C language features, and mainly exercise the optimization algorithms in compilers but not the front end of compilers that handles lexing and parsing. By contrast, existing DLGs cannot warrant well-defineness or even compilableness, and may use arbitrary language features depending on the programs in the dataset. Therefore, it is not surprising to see that LSG-generated programs trigger lower code coverage in compilers than DLG-generated programs, especially code coverage in the front end of compilers; in terms of bug detection ability, it is also expected that LSGs such as Csmith do not trigger more bugs than DLGs because LSGs have been heavily used by various compiler communities in the past (*e.g.*, Csmith has been used for years on a daily basis) and their bug detection ability has saturated. Overall, using LSGs as baselines to evaluate DLGs likely leads to biased conclusions.

Kitten. To help researchers fairly evaluate DLGs with proper baselines, we propose Kitten, a simple yet strong, fair baseline. Kitten is a *language-agnostic* program generation technique that supports abundant language features. Same as a DLG, Kitten requires a dataset consisting of programs. However, instead of the time-consuming step of training a deep neural model to create a DLG, Kitten directly generates new test programs by mutating the programs in the dataset. Specifically, given a program P in the dataset, Kitten randomly mutates the tokens of P or nodes in the parse tree [3] of P, and outputs the mutation result as a new test program.[2] Kitten has obvious advantages over both LSGs and DLGs. Kitten is much easier to implement than LSGs as it does not need domain

[2]Parsing a program into a list of tokens or a parse tree can be easily done with a parser generator such as Antlr [5], and the grammars of most main-stream programming languages are also available online [6].

knowledge to carefully design the rules to generate programs; Kitten does not have the constraints defined by Csmith or similar work [102], and thus supports diverse language features to comprehensively test compilers. Compared to DLGs, Kitten does not need to train a deep learning model, thus saving significant time and computational resources for training and generation; moreover, Kitten is interpretable, and extensible to support new mutation strategies.

Re-evaluating DLGs Using Kitten. Using Kitten as the baseline, we conducted a comprehensive experiment to empirically revisit the performance of two representative DLGs, *i.e.*, DeepSmith [37] and DeepFuzz [100]. Considering the simplicity of Kitten, it is reasonable to expect that DLGs should at least perform similarly to Kitten. However, with 1,500-CPU/GPU-hour experiment and analysis, the results show that the performance of existing DLGs is far from this simple, reasonable objective in three perspectives: bug detection ability, the diversity of language features in the generated programs and code coverage of compilers. In 72-hour testing on GCC, DeepSmith triggers 3 hang bugs and 1 distinct crashes, while Kitten triggers 1,750 hang bugs and 34 distinct crashes. The generated programs by Kitten cover 21,053 more lines and 26,853 more branches than the dataset, which is at least 2x as the one generated by DeepFuzz and DeepSmith. Moreover, the numbers of features leveraged in the generation by DeepSmith and DeepFuzz are also only around 64% of the one by Kitten. We believe that DLGs still have much room for improvement to compete against the simple baseline Kitten.

Contributions. We make the following contributions.

1. We identify that the evaluations of the state-of-the-art DLGs are biased due to the use of improper baselines.
2. We propose Kitten, a simple yet strong language-agnostic program generator as a fair baseline for evaluating DLGs.
3. We empirically demonstrate that DLGs have much room for improvement as they fail to compete with Kitten.
4. We make Kitten publicly available at `https://doi.org/10.5281/zenodo.7946826` to benefit future research on DLGs.

5.2 Preliminary

5.2.1 Compilers and Compiler Bugs

Given a program P, a compiler translates the source code of P into binary code, so that P can be executed by a computer. A typical compilation process consists of three stages. First, the front end of the compiler parses the source code and builds an intermediate representation of P. Second, the middle end performs various optimizations like dead code elimination on the intermediate representation to make P run faster and use fewer resources. Lastly, the back end converts the optimized internal representation into binary code. Bugs can occur in any of the aforementioned stages in a compiler [202, 165, 102, 85, 164, 86]. There are three major types of compiler bugs:

Crash. A compiler crashes, if it aborts the compilation process due to an error inside the compiler when compiling a program. For example, if a segmentation fault occurs when GCC is compiling a program, it aborts with an error message `internal compiler error: Segmentation fault`.

Hang. A hang (timeout) is a compiler bug when the compiler runs indefinitely to compile a program.

Miscompilation. Given a well-defined program P, a compiler miscompiles P, if the compiled binary of P is not semantically equivalent to P and thus behaves differently from P. This type of compiler bugs is referred to as miscompilation.

5.2.2 Program Generation for Compiler Testing

Automated compiler testing uses a program generator to automatically generate random test programs, and checks whether the compiler under test can correctly compile the generated programs. We categorize program generators into the following two classes.

LSG: Language-Specific Program Generator. LSGs generate programs by following a set of language-specific rules that are meticulously crafted by human experts based on the domain knowledge of the language under test. Csmith [202] is one representative LSG. It generates well-defined C programs that conform to the C language specification [71], in a top-down manner from a translation unit, functions and statements down to expressions. Each time a language construct is being generated, Csmith applies the generation rules

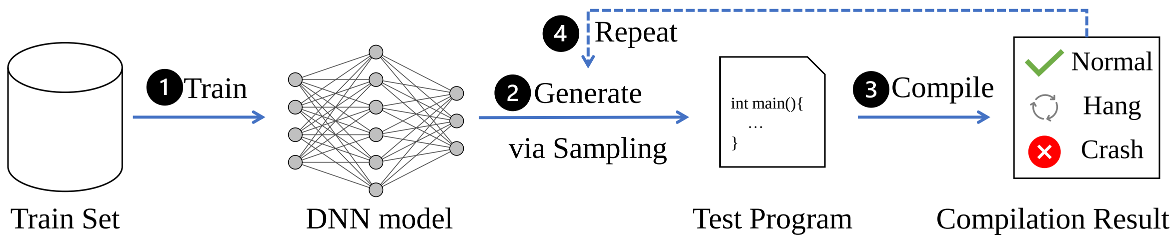


Figure 5.1: The workflow of using a DLG for testing a compiler.

to ensure that the generated program is compilable and well-defined. Several following studies extend this idea to other languages, such as OpenCL [93] and CUDA [72].

Despite the success of LSGs in finding compiler bugs, one major limitation of LSGs is that engineering a LSG requires language-specific knowledge to design generation rules and strategies. For example, to adapt the idea of Csmith to a new language, developers need to comprehend each feature of the new language and engineer the corresponding generation rules to ensure that the generated programs are compilable. Considering the complexity of programming languages, such work is labor-intensive and time-consuming, and the generation rules cannot be easily generalized to other languages. Further, it is challenging to take into consideration all language features and their possible combinations when designing generation rules. In fact, most LSGs only support a subset of language features and thus the expressiveness of the generated programs is rather limited.

DLG: Deep Learning-Based Program Generator. To overcome the limitation of LSGs, researchers proposed to use deep learning techniques to automatically learn a program generator from existing programs. Figure 5.1 shows the overall workflow of learning a DLG and applying the DLG to find compiler bugs. From a given dataset consisting of human-written programs, ① a deep learning model is trained as a DLG, which automatically learns the syntactical and semantic features of the language. Specifically, the model encodes the probability of the next token given a sequence of tokens as a prefix. ② The trained model generates a test program by iteratively querying the model to compute the next token. Specifically, starting from a prefix like `int main`, the DLG samples the subsequent token from the probabilities encoded in the trained model, until some termination tokens are generated. Then ③ the generated test program is fed to compilers under test and the compilation result is checked to see if any crash or hang bug is triggered. ④ This process is repeated until the time limit is reached or a sufficient number of test programs are generated. DeepSmith [37] and DeepFuzz [100] are the representative DLGs, which utilize Long Short-Term Memory (LSTM) model for training and generation. DSmith [199]

Property	DLGs	LSGs	Kitten
Need a set of programs for generation	✓	✗	✓
Language-agnostic	✓	✗	✓
Use arbitrary language features	✓	✗	✓
Warrant well-defineness/compilableness	✗	✓	✗
Can detect crash and hang bugs	✓	✓	✓
Can detect miscompilation bugs	✗	✓	✗

Table 5.1: Comparison of DLGs, LSGs, and Kitten.

and TSmith [198] also have similar workflows.

Different from LSGs, DLGs can only be used to find compiler crashes and hang bugs, but not miscompilation bugs. This is because the DLG-generated programs are not guaranteed to be compilable or free of undefined behaviors, and it is difficult to automatically determine whether the generated programs are free of undefined behaviors [202]. Table 5.1 summarizes the differences between DLGs and LSGs.

5.3 Revisiting the Evaluation of Deep Learning-Based Program Generators

As aforementioned, it is unfair to evaluate DLGs using LSGs and the resulting conclusions are likely to be biased. Thus, we aim to re-evaluate the performance of DLGs using a fair baseline. This section describes the design of our reevaluation experiment, and Kitten, a new fair baseline for evaluating DLGs.

5.3.1 Evaluation Methodology

To evaluate the performance of a program generator for compiler testing, we invoke it to continuously generate random programs for a certain period. After the generation, we feed them into a compiler under test for compilation. Finally, we measure the performance of this program generator from three perspectives: compiler bug detection ability, diversity of the language features in the generated programs, and code coverage. Each perspective and its metrics are introduced later in the corresponding sections. Since some measurements

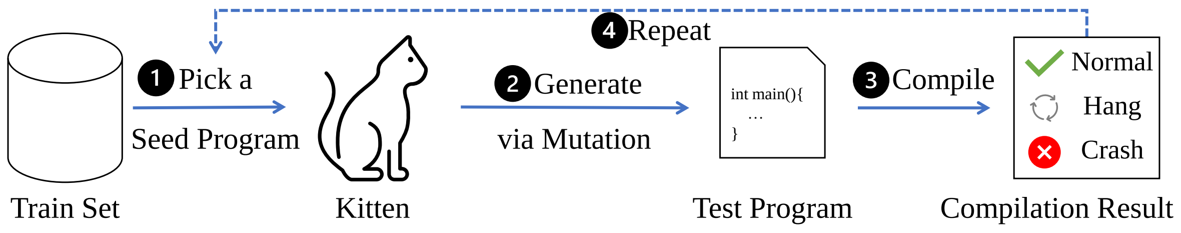


Figure 5.2: The workflow of using Kitten for testing a compiler.

may affect the compiler's efficiency, all the measurements are conducted after the program generation is finished. For example, measuring code coverage requires instrumenting compilers, resulting in a longer compilation time of the generated programs.

5.3.2 Selected Program Generators

We selected two representative DLGs from literature—DeepFuzz and DeepSmith—and re-evaluated their performance. They are selected for two reasons. First they were published in top-tier conferences of both software testing and artificial intelligence, representing the state of the art of DLGs. Second, their implementations are publicly available, avoiding re-implementation and thus minimizing threats to validity. Other work [198, 199] is not selected since their implementations are not publicly available. For each program generator, we followed their documentation and tried our best efforts to reproduce their results. However, we found that the performance of DeepFuzz is not comparable to the one mentioned in its publication. Similar issues are also raised by other developers in its repository [40]. Nevertheless, since DeepFuzz and DeepSmith have very similar workflow and architecture, we believe DeepSmith is a reasonable representative DLG work. We also include Csmith in our experiment, one of the most commonly used LSGs for C compilers.

To fairly evaluate the performance of DLGs, a proper baseline is necessary. Specifically, we are looking for a baseline that is similar to DLGs but much simpler than DLGs. Since such a baseline does not exist in the literature, we propose a new, fair and simple baseline, Kitten. Kitten is similar to DLGs except not using a DNN. Considering its simplicity, we expect that DLGs should at least perform similarly to Kitten.

5.3.3 Kitten: A New, Fair and Simple Baseline

Overview. Figure 5.2 shows the program generation workflow of Kitten. Unlike LSGs, Kitten shares many similarities with DLGs. Kitten takes as input a dataset of programs and outputs a new set of programs iteratively. For each iteration, it ① randomly takes one seed program and ② applies a random mutation operation to the seed to create a new program. After the program generation, ③ these programs are fed into a compiler under test. If a crash or hang bug is triggered, a potential bug for this compiler is detected. ④ The above process is repeated until the time limit is reached or a sufficient number of test programs are generated.

The entire program generation process of Kitten is language-agnostic, especially the mutation operators. Kitten supports diverse mutation operators and it is easy to integrate others. We implemented two types of operators from literature [8, 180], *i.e.*, tree-level and token-level mutations. Please note that the mutation operators of Kitten are language-agnostic and syntax-based, while the generation rules in LSGs are language-specific and semantic-based. The rules in LSGs require significantly more domain knowledge to design and implement than the mutation operators in Kitten.

Comparison with DLGs. Given a dataset of programs, DLGs leverage deep learning models to learn the syntactic and semantic rules and generate new programs, while Kitten generates random programs by directly mutating the programs in the dataset. Other than this, DLGs and Kitten share many commonalities as shown in Table 5.1: 1) generate new programs from a set of programs; 2) language-agnostic; 3) may use arbitrary language features ; 4) do not warrant well-defineness or even compilableness of the generated programs; 5) focus on detecting crash and timeout bugs in compilers.

Tree-Level Mutation

Parse Tree. A parse tree is a tree representation of program's syntactical structure. Figure 5.3 shows a program and its parse tree. There are two types of nodes in a parse tree, *i.e.*, non-leaf nodes (black rectangles in Figure 5.3) and leaf node (red circles in Figure 5.3). The former one refers to a non-terminal symbol in language grammar, *e.g.*, `expression` and `statement`. The latter one refers to a terminal symbol that cannot have child nodes [16]. Parsing programs into parse trees can be easily done with a parser generator (like Antlr [5]) and corresponding grammar [6], both of which are generally available for common programming languages.

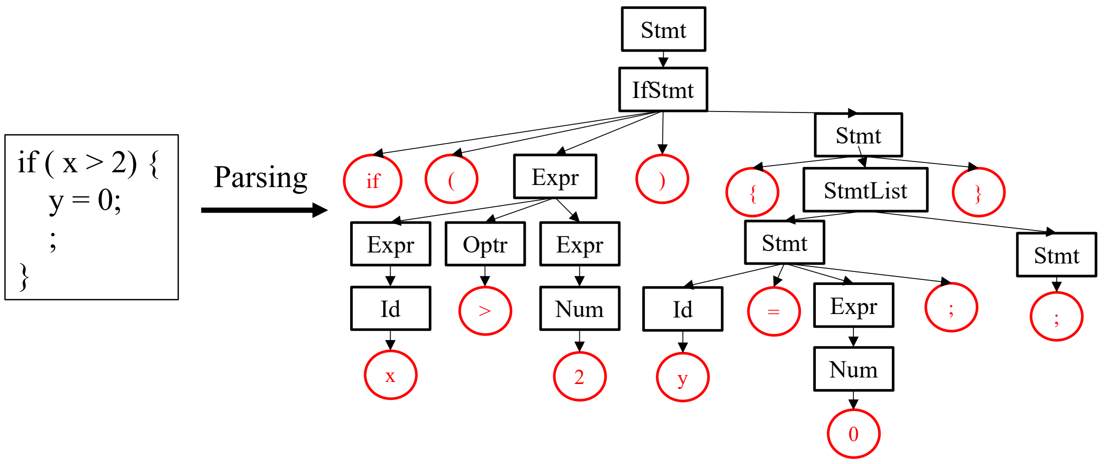


Figure 5.3: An example of a parse tree.

A tree-level mutation operator parses a seed program p to a parse tree t and mutates a sub-tree of t to generate a new program. The operator carefully checks the type of tree nodes and ensures the syntactic validity of the generated programs. In Kitten, we implemented three tree-level mutation operators [8, 180] and they are illustrated in Figure 5.4. Intuitively, these mutations randomly replace one sub-tree st of the parse tree t with a new sub-tree st', under the constraint that the root nodes of st' and st represent the same non-terminal symbol, such as an `expression` and `statement`, The major difference among these mutations is the source of the new subtree st'. In *Sub-tree Replacement*, st' is randomly generated by Kitten using the language grammar. *Sub-tree Splicing* randomly selects a sub-tree from other programs in the dataset. *Recursive Sub-tree Repeat* attempts to find a st' in the ancestor of st, *i.e.*, st is a sub-tree of st', and then uses st' to replace st arbitrary times.

Token-Level Mutation

Token-level mutation operations tokenize a seed program into a list of tokens and directly mutate this list. We implemented three types of token-level mutation in Kitten, namely *insertion*, *deletion* and *replacement*. As their name implies, these mutations randomly insert, delete or replace a token in the list. Table 5.2 shows the examples of the three mutations. Different from tree-level mutations, token-level mutations do not guarantee that the produced mutant is syntactically correct. This is intended since syntactically incorrect programs can find bugs in the front end of compilers.

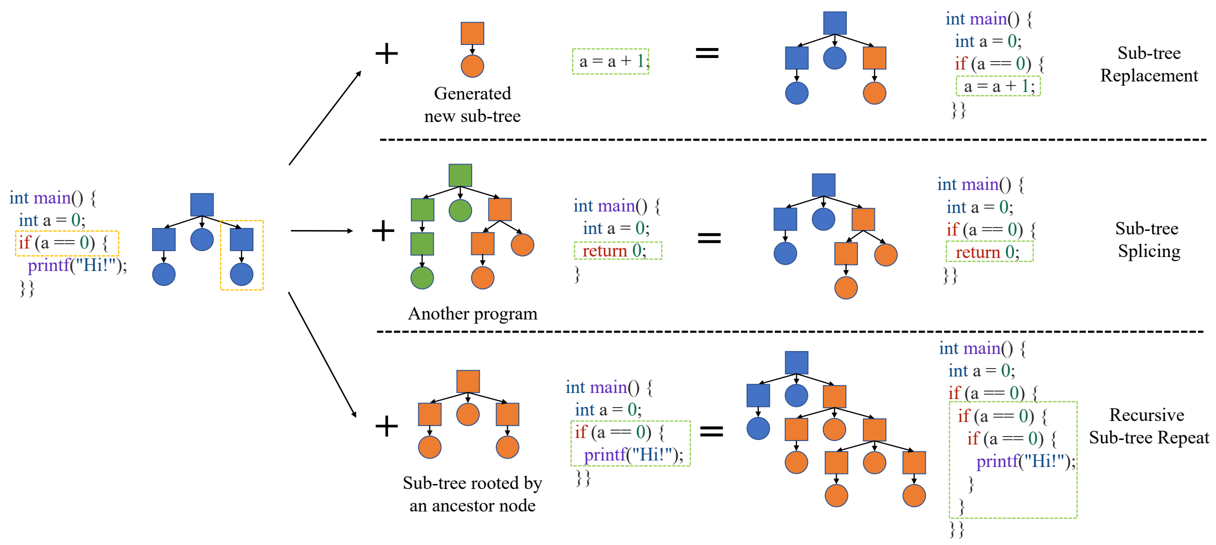


Figure 5.4: Tree-level mutations. The parse trees are simplified for illustration, thus not exactly matching the code examples.

	Program
Seed	`int main(){int a = 1 * 2;}`
Insertion	`int main(){int a = 1 * 2; int}`
Deletion	`int main(){a = 1 * 2;}`
Replacement	`int main(){float a = 1 * 2;}`

Table 5.2: Examples of token-level mutations.

5.3.4 Miscellaneous

Compiler and Dataset. We use all program generators to test GCC, one of the most commonly used and tested C compilers. Following existing DLGs [37, 100], we constructed a dataset using all the C files of the testsuite of GCC 11.2. Kitten directly used this dataset to generate new programs, while DeepSmith and DeepFuzz are trained using this dataset until the loss is saturated.

Platform and Duration. To ensure each generator has the same computation resources, each generator is deployed on a unique GPU virtual machine on a cloud platform with the same configuration. The longest duration used by prior work is 48 hours in

Generator	Average	Standard Deviation
DeepFuzz	138.54	9.78
DeepSmith	315,919.06	1,947.83
Csmith	2,021.32	187.72
Kitten	3,001,079.25	548,262.38

Table 5.3: The number of programs generated per hour.

DeepSmith. We choose a longer duration, *i.e.* 72 hours, for a comprehensive evaluation.

Noted that the program generation efficiency is not the primary concern in compiler testing, while effectiveness is more important. Nevertheless, we show the number of programs generated per hour in Table 5.3 for reference. DeepSmith and DeepFuzz have significantly different generation speeds. Such difference may be due to the aforementioned implementation issues in DeepFuzz. Csmith takes more effort than DLG and Kitten to ensure the semantic correctness of generated programs, resulting in a lower generation speed. Kitten is a better baseline than Csmith for evaluating DLGs since DLGs even do not guarantee compilableness of generated programs. Kitten has a much higher generation speed (9.5x at least) than DLGs since it directly mutates the parse tree or toke list, which requires less computation resource.

5.4 Empirical Findings

This section presents the empirical findings and discusses their implications.

5.4.1 Bug Detection Ability

Bug detection ability is one of the most important metrics to measure the quality of generated programs. In this RQ, we measured the number of bugs triggered by the program generated by each generator. We fed all the programs into the latest GCC development version (commit id `gf7a3ab`) and measured the number of crash and hang bugs. Specifically, for crash bugs, we first leveraged program reduction tool Perses [166] to reduce the bug-triggering program to the smallest amount of code that still replicates the bug. Second, we analyzed the stack traces and error messages of bugs. For bugs that have similar stack traces and error messages, we clustered them into one group and then investigated them

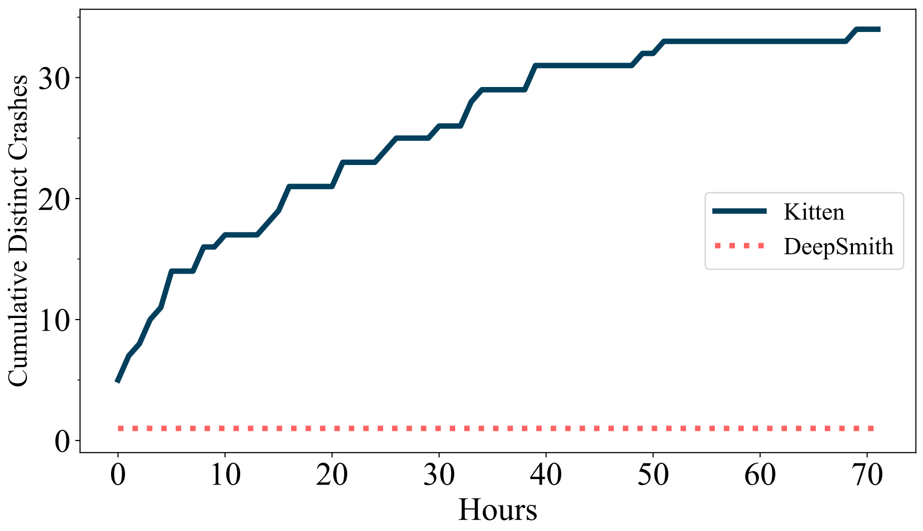


Figure 5.5: Distinct crashes. DeepFuzz and Csmith are not shown in this figure since they do not trigger any crash bugs.

manually. For hang bugs, as compilers do not throw error messages when hanging, there is no automatic way to deduplicate them and thus we only counted the total numbers. We did not measure the miscompilation bugs since it is a research challenge to automatically detect such bugs [85, 102] and neither DLGs nor Kitten is designed to find miscompilations.

Distinct Crash. Figure 5.5 shows the cumulative number of distinct compiler crashes triggered by Kitten and DeepSmith. In total, the programs generated by DeepSmith triggered 181 crash bugs but they all have the same root cause. In other words, DeepSmith only found one distinct crash bug. The programs generated by Kitten trigger 2,354 crashes and 34 of them are distinct bugs, which significantly outperforms DeepSmith. DeepFuzz and Csmith did not find any crash bugs.

Hang. Following the practice in compiler testing [202, 85], we used 120 seconds as the timeout threshold. DeepSmith triggered 3 hang bugs in GCC in 72 hours while Kitten triggered 1,750 hang bugs. DeepFuzz and Csmith do not trigger any hang bug.

We reduced the bug-triggering programs using Perses [166] and reported the discovered bugs to GCC after excluding the bugs that have already been reported recently. Figure 5.6 shows four of the bugs found by Kitten. All four bugs are new, and confirmed by GCC developers. Bug 105555 has been present since at least GCC 4.8.0, and was not found by any testing technique for over nine years. Bug 105554 has been fixed recently. Given that

```c
void foo() { __asm__("" :: "m"(({if(8);})));}
```

(a) GCC-100501, Crash

```c
static a(); b(void) {sizeof ( int [ a}
static c(); d(void) {sizeof((int[c
```

(b) GCC-104764, Hang

```c
__attribute__((target_clones(
"arch=core-avx2", "default")))
a(__attribute__((__vector_size__(
4 * sizeof(long)))) long) {}
```

(c) GCC-105554, Crash

```c
a() { &__imag *(_Complex *)a
```

(d) GCC-105555, Crash

Figure 5.6: Bugs found by DFLARE, including their bug-triggering programs, bug ids, and symptoms.

Kitten only takes three days to find four confirmed bugs, Kitten is an effective baseline in terms of bug detection.

Compared to Csmith, it seems that DLGs have a good bug detection ability since they find more bugs than Csmith. However, such an advantage is due to the unfair comparison since Csmith has been integrated into the daily testing of GCC and any triggered issues are expected to be fixed in development already [102]. In contrast, the limitation of DLGs in bug detection is demonstrated when Kitten is the evaluation baseline: the number of bugs detected by DLGs is far less than the one of Kitten, despite Kitten is a simple baseline without using complicated DNN models.

Finding and Advice 1: The bug detection abilities of DLGs are limited and did not outperform Kitten, a simple baseline without using DNN models. Future research should improve the bug detection abilities of DLGs to outperform Kitten.

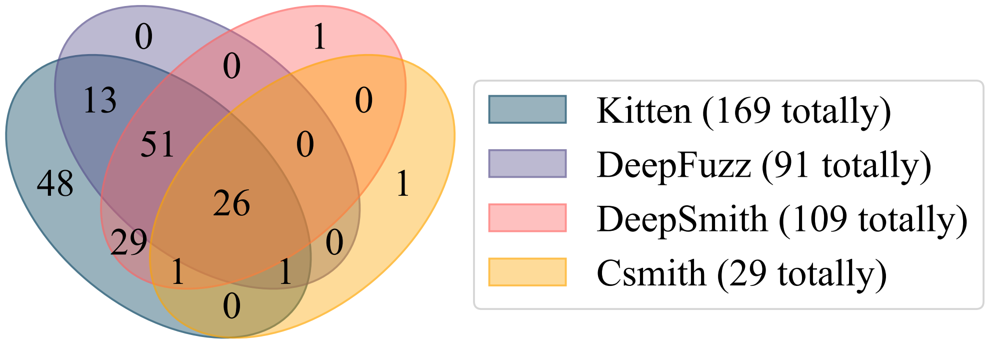


Figure 5.7: The AST node types of the programs generated by each generator.

5.4.2 Diversity of Language Features

To comprehensively test compilers, the generated program should cover a diverse set of language features. Following an existing study [44], we use the number of AST node types as a proxy metric of language features. Abstract Syntax Tree (AST) is a tree representation produced by compilers in the compilation. Each AST node type is regarded by compilers as a distinct type of component defined in language grammar, such as the variable declaration, function call, *etc.* For each generator, we sampled all the programs generated at first, twenty-fifth and forty-ninth hours and counted the number of distinct AST node types.

Figure 5.7 presents the result in a Venn diagram. The programs generated by DeepFuzz and DeepSmith contain 91 and 109 types of AST nodes, respectively. Both outperform Csmith which only utilizes 29 node types. This is because the generation rules embedded in Csmith only use a subset of language features to ensure the semantic correctness of the generated programs. Including additional features requires the developers manually design complex rules to guarantee correctness, which is a challenging task. As a result, it is unfair to compare DLGs with Csmith since DLGs do not explicitly care about such correctness of programs.

Kitten is a much fairer baseline for DLGs than Csmith, since neither DLGs nor Kitten limits the features they can use in program generation. The programs generated by Kitten have the largest number of distinct AST node types, *i.e.* 169. Moreover, all the AST node types in DeepFuzz and 108 out of 109 types in DeepSmith are included by the programs generated by Kitten. Based on this result, we may conclude that DLGs do not fully learn and leverage the language features in the training set. A possible reason is that, when sampling the next token, DLGs prefer the token that has a high likelihood according to cooccurrence. However, DLGs have limited knowledge of the syntactic and semantic meaning of tokens and fail to generate the programs toward diversity.

Finding and Advice 2: Existing DLGs do not fully utilize the language features in program generation, which may affect their bug detection abilities. Future research should encourage DLGs to use more diverse language features.

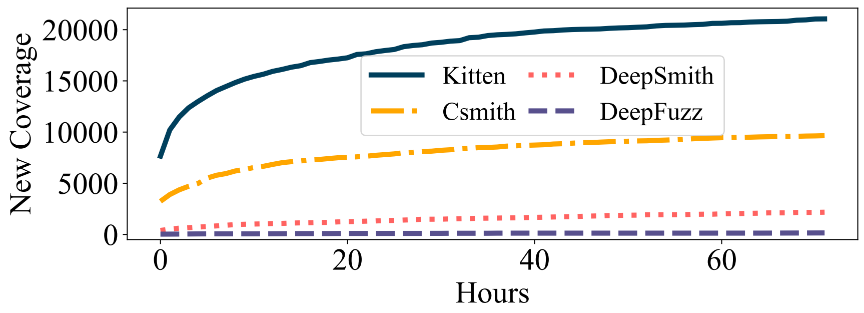
5.4.3 Code Coverage

Code coverage is an important metric in compiler testing [102, 164, 85]. It measures the code in compilers that is executed when compiling programs. High code coverage typically indicates that diverse code logic paths in compilers are exercised. For each set of generated programs, we fed them into GCC and used LCOV to measure the code coverage using three commonly used metrics: line, branch and function coverage.

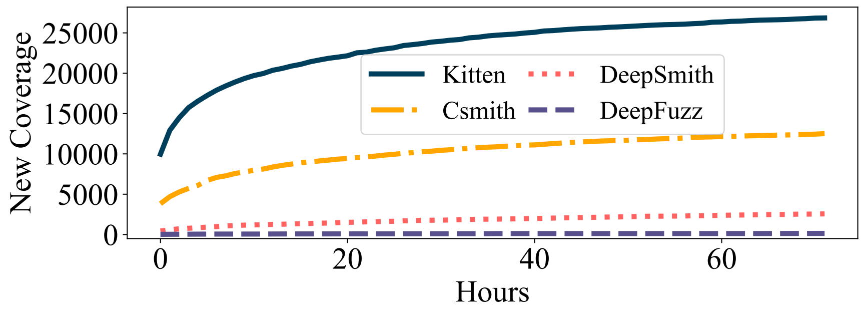
Figures 5.8a to 5.8c shows the number of lines, branches and functions covered by the programs generated by each generator but not by the seed programs. DeepSmith covers 2,175 lines, 2,562 branches, and 17 functions that are not covered by the seed programs, while Csmith covers 9,650 lines, 12,510 branches, and 206 functions. Even though DeepSmith generated much more (156x) programs and leveraged more (109 vs 29) features than Csmith, the code coverage achieved by DeepSmith is much lower than Csmith.

To understand the reason, we carefully investigated the difference between their coverage. Specifically, we analyzed how many new covered branches are in the front end of GCC and how many of them are in the middle/back end. Figure 5.9 shows the results. It is clear that the programs generated by DeepSmith mainly cover new branches in the front end of compilers, such as lexer and parser. By contrast, most of the branches covered by programs generated by Csmith are located in the middle/back end of compilers. Although the programs generated by Csmith only include limited language features, they can trigger complex optimization in the middle/back end of compilers, resulting in high code coverage. This result also demonstrated the effectiveness of language-specific rules designed by experts in compiler testing.

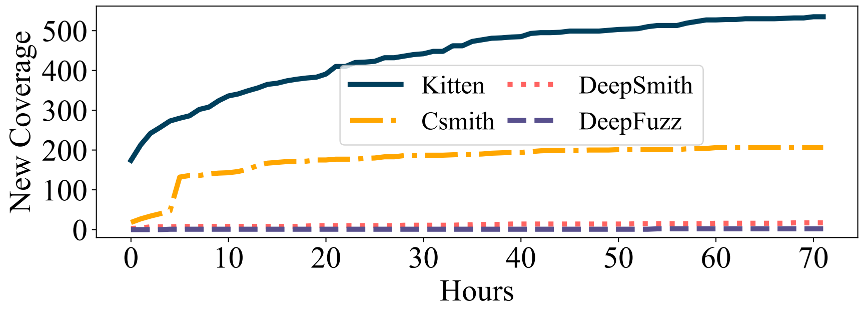
The line, branch, and function coverage achieved by Kitten is 9.68x, 10.48x and 31.47x as DeepSmith, respectively. As shown in Figure 5.9, such improvements locate in both the front end and middle/back end of compilers. Note that the advantage over DeepSmith is not only because of program generation speed. Furthermore, the programs generated by



(a) Line Coverage



(b) Branch Coverage



(c) Function Coverage

Figure 5.8: Code coverage of the programs generated by Kitten, Csmith, DeepSmith and DeepFuzz.

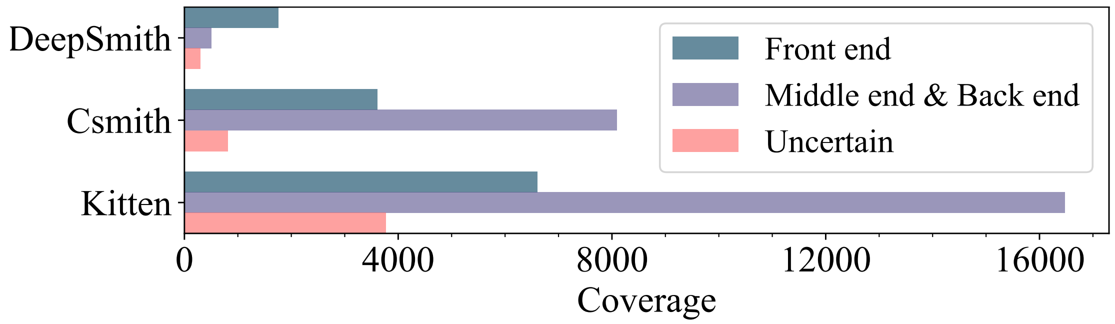


Figure 5.9: Distributions of new branch coverage. Uncertain refers to the cases of which category cannot be determined.

DeepSmith after seventy-two hours achieved lower coverage than the programs generated by Kitten in the first hour, despite DeepSmith generating more programs in those seventy-two hours than Kitten in the first hour.

Finding and Advice 3: DLGs does not outperform LSGs and Kitten in terms of code coverage. One of the reasons is that the programs generated by DLGs do not trigger diverse optimizations in compilers, which can be a promising research direction.

5.4.4 Implications

Our experiment results show that DLGs, such as DeepSmith and DeepFuzz, do not achieve outstanding performance in compiler testing. They have certain advantages over Csmith but such advantages are due to the unfair comparison as we mentioned in §5.2.2. Our experiment also shows that once the evaluation baseline is switched to Kitten, the performance of DLGs is worse than Kitten in all three perspectives: bug detection ability, diversity of language features and code coverage. Although DLGs attempt to leverage DNN models to learn the information of programs from datasets and generate new programs for compiler testing, it turns out that the results are not as good as the approach of Kitten, *i.e.*, directly mutating the programs in datasets.

We believe Kitten can benefit future DLG-related research in various aspects. First, Kitten sets a new fair baseline for DLGs. Future work of DLGs should at least have a significant improvement over Kitten. Second, the fact that Kitten outperforms existing DLGs may enlighten future research. Their limited performance implies that a simple end-to-end deep learning approach without incorporating any domain-specific information has not achieved decent performance in compiler testing. A promising research direction is to incorporate domain-specific knowledge extracted from the programs, such as parse trees, control-flow graphs and data-flow graphs, into DNN models. Moreover, as Kitten itself is shown to be effective, future research may study how to improve the workflow of Kitten using machine learning techniques.

5.5 Discussion

5.5.1 Experiment Setting

Please note that we choose to control the time and computation resources since it is a common practice in software testing to evaluate testing techniques by setting a limit on time and computation resources [214, 79]. For example, Google FuzzBench deploys each fuzzer for 24 hours. We believe this practice is related to how the software is tested in the industry, *i.e.*, developers need to adequately test the software before shipment date using the limited computation resources available to them.

We did not control the number of generated programs since the number of programs is easy to be tampered with. For example, one may easily merge multiple programs outputted by DFLARE into one program or vice versa. Moreover, even if DLGs, LSGs, and DFLARE are evaluated using the same number of generated programs, the conclusion of our empirical study remains the same. For bug detection ability, Figure 5.5 demonstrates that the number of crash bugs found by DFLARE in the first hour is more than the number of crash bugs found by DeepSmith in 72 hours, although the number of programs generated by DFLARE in the first hour is smaller than the number of programs generated by DeepSmith in 72 hours. As for the diversity of language features, DFLARE covers 168 AST node types in the first hour, while DeepSmith covers 109 AST node types in 72 hours. The results related to code coverage have been discussed in §5.4.3.

5.5.2 Future Work

There are several future work directions. First, our study can be generalized to other DL applications related to software testing and software engineering, such as code clone detection [184, 185] and vulnerability detection [92, 219, 47]. Finding the limitation of these DL applications may benefit future research. Second, to fairly evaluate DLGs, Kitten intentionally adopts the simple mutation operators and the random strategy for mutation operators. A promising direction is to improve Kitten by integrating novel mutation operators [200, 204, 162] and effective mutation operator schedulers [103]. Moreover, algorithms such as genetic algorithm [11, 188] and reinforcement learning [74] may also improve the effectiveness and efficiency of Kitten. Third, future research may improve the DLGs by encouraging DLGs to use more diverse language features and to trigger more optimizations in compilers, as discussed in §5.4.

5.6 Related Work

5.6.1 Compiler Testing

Compiler testing has been actively explored for many years. A common approach is automatically generating test programs with a generator and checking whether compilers properly compile these programs. As mentioned in §5.2.2, LSGs [202] and DLGs [100, 37] are two main kinds of generators. Kitten is much easier than LSG to implement as it does not require expert knowledge and massive engineering efforts to ensure the validity of generated programs. Different from DLGs that train DNN models using a given dataset, Kitten generates new programs by randomly mutating the ones in the dataset.

Another mainstream of compiler testing is detecting miscompilation bugs. For example, EMI [85, 164] generates a set of mutated programs that are equivalent to each other *w.r.t.* a set of inputs. After these programs are compiled, if the binary programs behave differently given these inputs, it indicates that at least one of them is miscompiled and a defect is detected. These approaches requires the insightful language-specific knowledge of developers to propose mutations that preserve the equivalence relationship *w.r.t.* a set of inputs, primarily targeting miscompilation bugs. In comparison, Kitten is a language-agnostic random program generator, targeting crash and hang bugs.

5.6.2 Revisiting Deep Learning Applications

There are several studies revisiting DL applications using simple baselines [113, 193, 192, 55, 138]. Mohammed *et al.* found that basic Recurrent Neural Networks with various heuristics can achieve similar performance with the state-of-the-art techniques in question answering tasks. Moreover, their study also found that the techniques without using DL have compatible performance with DL-based approaches. Goyal *et al.* studied several factors that are irrelevant to DL models and found that these factors have significant effects on the performance of DL applications in the task of classifying point cloud shapes. They also proposed a simple baseline that achieved similar or even better results than sophisticated DL techniques. Qian *et al.* conducted a similar analysis in the task of point cloud understanding. In summary, these studies focus on various DL applications, while our study is the first one to revisit DLGs in compiler testing. Kitten, the simple non-DL baseline proposed by us significantly outperforms DLGs.

5.7 Chapter Conclusion

This study argues that the evaluations of the DLGs are biased due to the improperly chosen baselines. LSGs are designed to utilize limited language features to generate well-defined programs, while DLGs generate arbitrary programs without concerns about the syntactic and semantic correctness. We revisited the evaluation of DLGs using Kitten, a fair, simple and strong baseline for DLGs. Instead of using DNN models, Kitten directly derives new programs from the dataset. Empirical results show that the advantage of DLGs claimed in their publications is likely due to the biased selection of baseline. Despite the simplicity of Kitten, DLGs cannot compete with Kitten in multiple metrics. With in-depth analysis of the evaluation results, we discuss potential directions for advancing future research on DLGs, and strongly believe that Kitten is the fair, right baseline for evaluating DLGs.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize this thesis and discuss several promising research directions for future work.

6.1 Summary

In summary, this thesis consists of three studies to help software developers to assess the reliability of DL applications. The reliability issues discussed in this thesis cover three important stages of the software development life cycle [38], *i.e.*, development, deployment, and assessment. This thesis **provided three techniques** to help software developers to assess the reliability of DL applications. The first study proposed a testing methodology to detect unreliable inferences in DL-based image classifications. The second work presented a novel testing technique DFLARE to find the deviated behaviors of compressed DL applications. The third work proposed a simple, fair, and strong baseline Kitten for DL-based program generator in compiler testing. With these techniques, this thesis **conducted comprehensive empirical studies** to understand the reliability issues from multiple perspectives, including their perseverance, impacts and so on. For example, the empirical study in the first study showed that such unreliable inferences are pervasive in the state-of-the-art DL-based image classifications. The third work revealed that the unreliable assessment of DLGs resulted in biased conclusions in the evaluation of DLGs. This thesis further **offered actionable suggestions and techniques** to help software developers to improve the reliability of DL applications. For instance, in the second work, it is demonstrated that the triggering inputs found by DFLARE can be used by DREPAIR to repair the deviated behaviors. The first and third study provided a collection of suggestions to benefit future research in image classification and DLGs, respectively.

6.2 Future Work

Besides the future work listed in the previous chapters, there are many promising research directions. First, besides image classifications and compiler testing, DL applications have been used in many areas. These applications may have domain-specific reliability issues [211, 21]. Future research may help developers to identify the reliability issues in domain-specific applications. For example, a recent study proposes an effective prompting methodology to increase the reliability of large language models like GPT [158]. Another study focuses on new evaluation metrics for NLP models other than accuracy [145]. Assessing the reliability issues of these DL applications usually requires domain-specific knowledge. It would facilitate the revealing process if such domain-specific knowledge can be automatically extracted.

Second, the reliability of DL applications is also affected by the infrastructures of DL, including DL frameworks, such as TensorFlow and PyTorch, DL compilers like TVM, low-level computation libraries, *e.g.*, NVIDIA CUDA, and the DL hardware like GPUs and TPUs. Effective methodologies to find the defects in such software and hardware can also improve the reliability of DL applications [156, 91, 187, 194].

Third, the techniques to improve the reliability of DL applications are also important to the community. Possible approaches in this direction include novel training and testing methodologies, innovative theory for software development and maintenance, and so on. A significant challenge is how to alleviate the side effect of these methodologies on the performance of DL applications. For example, adversarial training has been demonstrated as one effective way to improve the adversarial robustness of DL applications [216, 155]. However, the accuracy of DL application after adversarial training is likely to be decreased [216]. Future research may work on improving the reliability of DL applications while preserving their performance.

References

[1] Waleed Abdulla. 2017. Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow. Retrieved May 16, 2023 from https://github.com/matterport/Mask_RCNN

[2] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black Box Fairness Testing of Machine Learning Models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 625–635. https://doi.org/10.1145/3338906.3338937

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, USA, Chapter 6. Intermediate-Code Generation.

[4] Naveed Akhtar and Ajmal Mian. 2018. Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey. *IEEE Access* 6 (2018), 14410–14430.

[5] Antlr. 2022. Antlr. Retrieved May 16, 2023 from https://www.antlr.org/

[6] Antlr. 2022. Antlr Grammar. Retrieved May 16, 2023 from https://github.com/antlr/grammars-v4

[7] ARM. 2022. *Image Classification with MobilenetV2, Arm NN, and TensorFlow Lite Delegate pre-built binaries Tutorial.* Retrieved May 16, 2022 from https://developer.arm.com/documentation/102561/2111

[8] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27,*

2019. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[9] Audiomentations. 2023. *Audiomentations: A Python library for audio data augmentation.* Retrieved May 16, 2023 from https://github.com/iver56/audiomentations

[10] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[11] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. 1998. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[13] Emanuel Ben Baruch, Tal Ridnik, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. 2021. *Repository of TResNet-L.* Retrieved May 16, 2021 from https://github.com/Alibaba-MIIL/ASL

[14] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.

[15] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. 2018. Practical Black-Box Attacks on Deep Neural Networks Using Efficient Query Mechanisms. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XII (Lecture Notes in Computer Science, Vol. 11216)*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer, 158–174. https://doi.org/10.1007/978-3-030-01258-8_10

[16] Bison. 2022. Bison Grammar Files. Retrieved May 16, 2022 from http://web.mit.edu/gnu/doc/html/bison_6.html#SEC34

[17] Charles Boncelet. 2009. Chapter 7 - Image Noise Models. In *The Essential Guide to Image Processing*, Al Bovik (Ed.). Academic Press, Boston, 143–167. https://doi.org/10.1016/B978-0-12-374457-9.00007-X

[18] Cristian Buciluundefined, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model Compression *(KDD '06)*. Association for Computing Machinery, New York, NY, USA, 535–541. https://doi.org/10.1145/1150402.1150464

[19] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=HylxE1HKwS

[20] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3510003.3510099

[21] Nicholas Carlini and David Wagner. 2018. Audio Adversarial Examples: Targeted Attacks on Speech-to-Text. In *2018 IEEE Security and Privacy Workshops (SPW)*. 1–7. https://doi.org/10.1109/SPW.2018.00009

[22] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 39–57. https://doi.org/10.1109/SP.2017.49

[23] Lahiru D. Chamain, Siyu Qi, and Zhi Ding. 2022. End-to-End Image Classification and Compression With Variational Autoencoders. *IEEE Internet of Things Journal* 9, 21 (2022), 21916–21931. https://doi.org/10.1109/JIOT.2022.3182313

[24] Tsong Yueh Chen, Shing-Chi Cheung, and Siu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report HKUST-CS98-01. Department of Computer Science, HKUST, Hong Kong.

[25] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (Jan. 2018), 27 pages. https://doi.org/10.1145/3143561

[26] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software*

Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

[27] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 85–99. https://doi.org/10.1145/2908080.2908095

[28] Zuohui Chen, RenXuan Wang, Yao Lu, jingyang Xiang, and Qi Xuan. 2021. Adversarial Sample Detection via Channel Pruning. In *ICML 2021 Workshop on Adversarial Machine Learning*. https://openreview.net/forum?id=MKfq7TuRBCK

[29] Minhao Cheng, Thong Le, Pin-Yu Chen, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. 2019. Query-Efficient Hard-label Black-box Attack: An Optimization-based Approach. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=rJlk6iRqKX

[30] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (apr 2021), 33 pages. https://doi.org/10.1145/3436877

[31] Jang Hyun Cho and Bharath Hariharan. 2019. On the Efficacy of Knowledge Distillation. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 4793–4801. https://doi.org/10.1109/ICCV.2019.00489

[32] François Chollet et al. 2015. Keras. Retrieved May 16, 2023 from https://keras.io

[33] François Chollet et al. 2015. Keras Applications. Retrieved March 3, 2023 from https://keras.io/api/applications/

[34] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. 2020. A Comprehensive Survey on Model Compression and Acceleration. *Artif. Intell. Rev.* 53, 7 (oct 2020), 5113–5155. https://doi.org/10.1007/s10462-020-09816-7

[35] William Gemmell Cochran. 1963. *Sampling techniques. 2nd edition.* John Wiley & Sons.

[36] Harald Cramer. 1946. *Mathematical methods of statistics.* Princeton University Press, Princeton.

[37] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[38] Alan M. Davis, Edward H. Bersoff, and Edward R. Comer. 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Trans. Software Eng.* 14, 10 (1988), 1453–1461. https://doi.org/10.1109/32.6190

[39] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[40] Developers. 2022. GitHub Issue for DeepFuzz. Retrieved July 26, 2022 from https://github.com/s3team/DeepFuzz/issues/1

[41] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[42] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. 2017. Validating a Deep Learning Framework by Metamorphic Testing. In *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*. IEEE Computer Society, 28–34. https://doi.org/10.1109/MET.2017.2

[43] Thanh-Nghi Doan and François Poulet. 2012. Large Scale Image Classification: Fast Feature Extraction, Multi-codebook Approach and Multi-core SVM Training. In *Advances in Knowledge Discovery and Management - Volume 4 [Best of EGC 2012, Bordeaux, France] (Studies in Computational Intelligence, Vol. 527)*, Fabrice Guillet, Bruno Pinaud, Gilles Venturini, and Djamel Abdelkader Zighed (Eds.). Springer, 155–172. https://doi.org/10.1007/978-3-319-02999-3_9

[44] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2023. Bash in the Wild: Language Usage, Code Smells, and Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 8:1–8:22. https://doi.org/10.1145/3517193

[45] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/3213846.3213858

[46] Hazem M. Fahmy, Fabrizio Pastore, Mojtaba Bagherzadeh, and Lionel C. Briand. 2021. Supporting Deep Neural Network Safety Analysis and Retraining Through Heatmap-Based Unsupervised Learning. , 1641–1657 pages. https://doi.org/10.1109/TR.2021.3074750

[47] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 508–512. https://doi.org/10.1145/3379597.3387501

[48] Christiane Fellbaum. 2006. WordNet(s). In *Encyclopedia of Language & Linguistics (Second Edition)* (second edition ed.), Keith Brown (Ed.). Elsevier, Oxford, 665–670. https://doi.org/10.1016/B0-08-044854-2/00946-9

[49] Yoav Freund and Robert E. Schapire. 1995. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, Paul Vitányi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–37.

[50] Karl Pearson F.R.S. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175. https://doi.org/10.1080/14786440009463897 arXiv:https://doi.org/10.1080/14786440009463897

[51] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. 2019. ImageNet-trained CNNs are biased towards

texture; increasing shape bias improves accuracy and robustness. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bygh9j09KX

[52] Rafael C. Gonzalez and Richard E. Woods. 2008. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J.

[53] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6572

[54] Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. 2021. Knowledge Distillation: A Survey. *International Journal of Computer Vision* 129, 6 (mar 2021), 1789–1819. https://doi.org/10.1007/s11263-021-01453-z

[55] Ankit Goyal, Hei Law, Bowei Liu, Alejandro Newell, and Jia Deng. 2021. Revisiting Point Cloud Shape Classification with a Simple and Effective Baseline. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 3809–3820. https://proceedings.mlr.press/v139/goyal21a.html

[56] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. BadNets: Evaluating Backdooring Attacks on Deep Neural Networks. *IEEE Access* 7 (2019), 47230–47244. https://doi.org/10.1109/ACCESS.2019.2909068

[57] Chuan Guo, Jacob R. Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Q. Weinberger. 2019. Simple Black-box Adversarial Attacks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 2484–2493. http://proceedings.mlr.press/v97/guo19a.html

[58] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 739–743. https://doi.org/10.1145/3236024.3264835

[59] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 810–822. https://doi.org/10.1109/ASE.2019.00080

[60] Manish Gupta and Puneet Agrawal. 2022. Compression of Deep Learning Models for Text: A Survey. *ACM Trans. Knowl. Discov. Data* 16, 4, Article 61 (jan 2022), 55 pages. https://doi.org/10.1145/3487045

[61] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[62] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *NIPS'15* (Montreal, Canada). MIT Press, Cambridge, MA, USA, 1135–1143.

[63] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep Speech: Scaling up end-to-end speech recognition. *CoRR* abs/1412.5567 (2014). arXiv:1412.5567 http://arxiv.org/abs/1412.5567

[64] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2980–2988. https://doi.org/10.1109/ICCV.2017.322

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[66] Jan Hendrik Hosang, Rodrigo Benenson, Piotr Dollár, and Bernt Schiele. 2016. What Makes for Effective Detection Proposals? *IEEE Trans. Pattern Anal. Mach. Intell.* 38, 4 (2016), 814–830. https://doi.org/10.1109/TPAMI.2015.2465908

[67] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[68] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Wei Ma, Mike Papadakis, and Yves Le Traon. 2022. Characterizing and Understanding the Behavior of Quantized Models for Reliable Deployment. *CoRR* abs/2204.04220 (2022). https://doi.org/10.48550/arXiv.2204.04220 arXiv:2204.04220

[69] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[70] Kun Huang, Bingbing Ni, and Xiaokang Yang. 2019. Efficient Quantization for Neural Networks with Binary Weights and Low Bitwidth Activations. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 473, 8 pages. https://doi.org/10.1609/aaai.v33i01.33013854

[71] IOS. 2005. ISO/IEC 9899:TC2: Programming Languages—C. Retrieved May 16, 2023 from https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[72] Bo Jiang, Xiaoyan Wang, W. K. Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. CUDAsmith: A Fuzzer for CUDA Compilers. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. 861–871. https://doi.org/10.1109/COMPSAC48688.2020.0-156

[73] James M. Joyce. 2011. *Kullback-Leibler Divergence*. Springer Berlin Heidelberg, Berlin, Heidelberg, 720–722. https://doi.org/10.1007/978-3-642-04898-2_327

[74] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[75] Robert E. Kass, Bradley P. Carlin, Andrew Gelman, and Radford M. Neal. 1998. Markov chain monte carlo in practice: A roundtable discussion. *American Statistician* 52, 2 (May 1998), 93–100.

[76] Faiq Khalid, Hassan Ali, Hammad Tariq, Muhammad Abdullah Hanif, Semeen Rehman, Rehan Ahmed, and Muhammad Shafique. 2019. QuSecNets: Quantization-based Defense Mechanism for Securing Deep Neural Network against Adversarial Attacks. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 182–187. https://doi.org/10.1109/IOLTS.2019.8854377

[77] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1039–1049. https://doi.org/10.1109/ICSE.2019.00108

[78] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. 2023. Segment Anything. arXiv:2304.02643 [cs.CV]

[79] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[80] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. 2017. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html* (2017).

[81] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2009. The CIFAR-10 dataset. (2009). Retrieved May 16, 2023 from http://www.cs.toronto.edu/~kriz/cifar.html

[82] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/

4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[83] Fred Lambert. 2016. *Understanding the fatal Tesla accident on Autopilot and the NHTSA probe.* Retrieved May 16, 2023 from https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/

[84] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174.

[85] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[86] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[87] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[88] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*. 2278–2324.

[89] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. Retrieved May 16, 2023 from http://yann.lecun.com/exdb/mnist/

[90] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJqFGTslg

[91] Meiziniu Li, Jialun Cao, Yongqiang Tian, Tsz On Li, Ming Wen, and Shing-Chi Cheung. 2023. COMET: Coverage-Guided Model Generation For Deep Learning

Library Testing. *ACM Trans. Softw. Eng. Methodol.* (feb 2023). https://doi.org/10.1145/3583566 Just Accepted.

[92] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 292–303. https://doi.org/10.1145/3468264.3468597

[93] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. (2015), 65–76. https://doi.org/10.1145/2737924.2737986

[94] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. 8693 (2014), 740–755. https://doi.org/10.1007/978-3-319-10602-1_48

[95] Tao Lin, Sebastian U. Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. 2020. Dynamic Model Pruning with Feedback. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJem8lSFwB

[96] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[97] Jing Liu, Bohan Zhuang, Zhuangwei Zhuang, Yong Guo, Junzhou Huang, Jinhui Zhu, and Mingkui Tan. 2022. Discrimination-Aware Network Pruning for Deep Model Compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4035–4051. https://doi.org/10.1109/TPAMI.2021.3066410

[98] Qi Liu, Tao Liu, Zihao Liu, Yanzhi Wang, Yier Jin, and Wujie Wen. 2018. Security analysis and enhancement of model compressed deep learning systems under adversarial attacks. In *23rd Asia and South Pacific Design Automation Conference, ASP-DAC 2018, Jeju, Korea (South), January 22-25, 2018*, Youngsoo Shin (Ed.). IEEE, 721–726. https://doi.org/10.1109/ASPDAC.2018.8297407

[99] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector.

In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9905)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[100] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[101] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/ARXIV.1907.11692

[102] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

[103] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[104] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 120–131. https://doi.org/10.1145/3238147.3238202

[105] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*, Sudipto Ghosh, Roberto Natella, Bojan Cukic, Robin Poston, and Nuno Laranjeiro (Eds.). IEEE Computer Society, 100–111. https://ieeexplore.ieee.org/xpl/conhome/8536838/proceeding

[106] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/3236024.3236082

[107] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. 2019. NIC: Detecting Adversarial Samples with Neural Network Invariant Checking. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss2019/

[108] Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.

[109] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[110] Sean Meyn and Richard L. Tweedie. 2009. *Markov Chains and Stochastic Stability* (2nd ed.). Cambridge University Press, USA.

[111] Asit K. Mishra and Debbie Marr. 2018. Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=B1ae1lZRb

[112] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., New York, NY, USA.

[113] Salman Mohammed, Peng Shi, and Jimmy Lin. 2018. Strong Baselines for Simple Question Answering over Knowledge Graphs with and without Neural Networks. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 291–296. https://doi.org/10.18653/v1/N18-2047

[114] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. 2019. Layer-wise relevance propagation: an overview. In *Explainable AI: interpreting, explaining and visualizing deep learning*. Springer, 193–209.

[115] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2574–2582. https://doi.org/10.1109/CVPR.2016.282

[116] Mozilla. 2020. *Augmentation*. Retrieved May 16, 2023 from https://deepspeech.readthedocs.io/en/r0.9/TRAINING.html#augmentation

[117] Mozilla. 2020. *Repository of DeepSpeech*. Retrieved May 16, 2023 from https://github.com/mozilla/DeepSpeech

[118] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.

[119] Nicolas M Müller and Karla Markert. 2019. Identifying mislabeled instances in classification datasets. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[120] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (Haifa, Israel) *(ICML'10)*. Omnipress, USA, 807–814. http://dl.acm.org/citation.cfm?id=3104322.3104425

[121] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 785–796. https://doi.org/10.1109/ASE.2019.00078

[122] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[123] NVIDIA. 2021. *Image Classification*. Retrieved May 16, 2023 from https://docs.nvidia.com/metropolis/TLT/tlt-user-guide/text/image_classification.html

[124] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *ICML 2019, 9-15 June 2019, Long Beach, California, USA*. 4901–4911.

[125] ONNX. 2022. ONNX Inference. Retrieved May 20, 2022 from https://onnxruntime.ai/

[126] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[127] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[128] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: differential verification of deep neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 714–726. https://doi.org/10.1145/3377811.3380337

[129] Brandon Paulsen, Jingbo Wang, Jiawei Wang, and Chao Wang. 2020. NEURODIFF: Scalable Differential Verification of Neural Networks using Fine-Grained Approximation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 784–796. https://doi.org/10.1145/3324884.3416560

[130] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 1–18. https://doi.org/10.1145/3132747.3132785

[131] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[132] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 771–783. https://doi.org/10.1145/3324884.3416545

[133] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=S1XolQbRW

[134] PyTorch. 2021. *PyTorch Examples*. Retrieved May 16, 2021 from https://github.com/pytorch/examples/tree/main/imagenet

[135] PyTorch. 2021. *Table of all available classification weights*. Retrieved May 16, 2021 from https://pytorch.org/vision/stable/models.html#table-of-all-available-classification-weights

[136] PyTorch. 2022. Models and pre-trained weights: Quantized models. Retrieved 2022-08-19 from https://pytorch.org/vision/stable/models.html#quantized-models

[137] Pytorch. 2022. Pytorch. Retrieved May 20, 2022 from https://www.pytorch.org

[138] Guocheng Qian, Yuchen Li, Houwen Peng, Jinjie Mai, Hasan Hammoud, Mohamed Elhoseiny, and Bernard Ghanem. 2022. PointNeXt: Revisiting PointNet++ with Improved Training and Scaling Strategies. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 23192–23204.

[139] Ge Qin, Bogdan Vrusias, and Lee Gillam. 2010. Background Filtering for Improving of Object Detection in Images. In *20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey, 23-26 August 2010*. IEEE Computer Society, 922–925. https://doi.org/10.1109/ICPR.2010.231

[140] Roshan Rao, Joshua Meier, Tom Sercu, Sergey Ovchinnikov, and Alexander Rives. 2021. Transformer protein language models are unsupervised structure learners. (2021). https://openreview.net/forum?id=fylclEqgvgd

[141] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 9908)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer, 525–542. https://doi.org/10.1007/978-3-319-46493-0_32

[142] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788. https://doi.org/10.1109/CVPR.2016.91

[143] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 6 (June 2017), 1137–1149. https://doi.org/10.1109/TPAMI.2016.2577031

[144] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 1135–1144.

[145] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4902–4912. https://doi.org/10.18653/v1/2020.acl-main.442

[146] Vincenzo Riccio and Paolo Tonella. 2020. Model-Based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 876–888. https://doi.org/10.1145/3368089.3409730

[147] Tal Ridnik, Emanuel Ben Baruch, Nadav Zamir, Asaf Noy, Itamar Friedman, Matan Protter, and Lihi Zelnik-Manor. 2021. Asymmetric Loss For Multi-Label Classification. In *2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021*. IEEE, 82–91. https://doi.org/10.1109/ICCV48922.2021.00015

[148] Danny Roobaert, Michael Zillich, and Jan-Olof Eklundh. 2001. A pure learning approach to background-invariant object recognition using pedagogical support vector learning. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, Vol. 2. II–II. https://doi.org/10.1109/CVPR.2001.990982

[149] Amir Rosenfeld, Richard S. Zemel, and John K. Tsotsos. 2018. The Elephant in the Room. *CoRR* abs/1808.03305 (2018). arXiv:1808.03305 http://arxiv.org/abs/1808.03305

[150] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (01 Oct 1986), 533–536. https://doi.org/10.1038/323533a0

[151] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2014. ImageNet Large Scale Visual Recognition Challenge. https://doi.org/10.48550/ARXIV.1409.0575

[152] Jorge Sánchez and Florent Perronnin. 2011. High-dimensional signature compression for large-scale image classification. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2011, Colorado Springs, CO, USA, 20-25 June 2011*. IEEE Computer Society, 1665–1672. https://doi.org/10.1109/CVPR.2011.5995504

[153] Rakesh Sankar, Ashish Tiwari, Ramsingh G, and Keerthi M K. 2022. *Image classification on QCS610 development kit*. Retrieved May 16, 2023 from https://developer.qualcomm.com/project/image-classification-qcs610-development-kit

[154] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 618–626. https://doi.org/10.1109/ICCV.2017.74

[155] Ali Shafahi, Mahyar Najibi, Mohammad Amin Ghiasi, Zheng Xu, John Dickerson, Christoph Studer, Larry S Davis, Gavin Taylor, and Tom Goldstein. 2019. Adversarial training for free! *Advances in Neural Information Processing Systems* 32 (2019).

[156] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[157] Yucheng Shi, Siyu Wang, and Yahong Han. 2019. Curls & Whey: Boosting Black-Box Adversarial Attacks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 6519–6527. https://doi.org/10.1109/CVPR.2019.00668

[158] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Lee Boyd-Graber, and Lijuan Wang. 2023. Prompting GPT-3 To Be Reliable. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=98p5x51L5af

[159] Taylor Simons and Dah-Jye Lee. 2019. A Review of Binarized Neural Networks. *Electronics* 8, 6 (2019). https://doi.org/10.3390/electronics8060661

[160] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[161] Pravendra Singh, Vinay Kumar Verma, Piyush Rai, and Vinay P. Namboodiri. 2019. Play and Prune: Adaptive Filter Pruning for Deep Model Compression. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 3460–3466.

[162] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 244–256. https://doi.org/10.1145/3460319.3464814

[163] Pierre Stock and Moustapha Cissé. 2018. ConvNets and ImageNet Beyond Accuracy: Understanding Mistakes and Uncovering Biases. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings,*

Part VI (Lecture Notes in Computer Science, Vol. 11210), Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer, 504–519. https://doi.org/10.1007/978-3-030-01231-1_31

[164] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[165] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 294–305. https://doi.org/10.1145/2931037.2931074

[166] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 361–371. https://doi.org/10.1145/3180155.3180236

[167] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. 2018. Attacks Meet Interpretability: Attribute-steered Detection of Adversarial Samples. 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., 7717–7728.

[168] TensorFlow. 2022. TensorFlow. Retrieved May 16, 2022 from https://www.tensorflow.org/

[169] TensorFlow. 2022. TensorFlow Lite. Retrieved May 20, 2022 from https://www.tensorflow.org/lite

[170] Nvidia TensorRT. 2022. Nvidia. Retrieved May 20, 2022 from https://developer.nvidia.com/tensorrt

[171] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang. 2021. To what extent do DNN-based image classification models make unreliable inferences? *Empir. Softw. Eng.* 26, 4 (2021), 84. https://doi.org/10.1007/s10664-021-09985-1

[172] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 303–314. https://doi.org/10.1145/3180155.3180220

[173] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, Macao, China, August 19-25, 2023*, Edith Elkind (Ed.). ijcai.org.

[174] Yongqiang Tian, Zhihua Zeng, Ming Wen, Yepang Liu, Tzu-yang Kuo, and Shing-Chi Cheung. 2020. EvalDNN: A Toolbox for Evaluating Deep Neural Network Models. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3377812.3382133

[175] Yongqiang Tian, Wuqi Zhang, Ming Wen, Shing-Chi Cheung, Chengnian Sun, Shiqing Ma, and Yu Jiang. 2023. Finding Deviated Behaviors of the Compressed DNN Models for Image Classifications. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 128 (jul 2023), 32 pages. https://doi.org/10.1145/3583564

[176] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2020. Testing DNN Image Classifiers for Confusion & Bias Errors. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1122–1134. https://doi.org/10.1145/3377811.3380400

[177] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2021. *Repository of DeepInspect: Testing DNN Image Classifier for Confusion & Bias Errors*. Retrieved May 16, 2021 from https://github.com/ARiSE-Lab/DeepInspect

[178] Florian Tramèr, Vaggelis Atlidakis, Roxana Geambasu, Daniel J. Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. FairTest: Discovering Unwarranted Associations in Data-Driven Applications. In *2017 IEEE European Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 401–416. https://doi.org/10.1109/EuroSP.2017.29

[179] Hanrui Wang, Jiaqi Gu, Yongshan Ding, Zirui Li, Frederic T. Chong, David Z. Pan, and Song Han. 2022. QuantumNAT: Quantum Noise-Aware Training with Noise Injection, Quantization and Normalization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3489517.3530400

[180] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[181] Jialai Wang, Han Qiu, Yi Rong, Hengkai Ye, Qi Li, Zongpeng Li, and Chao Zhang. 2022. BET: Black-Box Efficient Testing for Convolutional Neural Networks. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 164–175. https://doi.org/10.1145/3533767.3534386

[182] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 8612–8620. https://doi.org/10.1109/CVPR.2019.00881

[183] Shuai Wang and Zhendong Su. 2020. Metamorphic Object Insertion for Testing Object Detection Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE 2020)*. ACM, New York, NY, USA, 1053–1065. https://doi.org/10.1145/3324884.3416584

[184] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[185] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020. Modular Tree Network

for Source Code Representation Learning. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 31 (sep 2020), 23 pages. https://doi.org/10.1145/3409331

[186] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 788–799. https://doi.org/10.1145/3368089.3409761

[187] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 995–1007. https://doi.org/10.1145/3510003.3510041

[188] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[189] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[190] Guoqiang Wu and Jun Zhu. 2020. Multi-label classification: do Hamming loss and subset accuracy really conflict with each other?. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[191] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 4820–4828. https://doi.org/10.1109/CVPR.2016.521

[192] Yu Wu, Lu Jiang, and Yi Yang. 2020. Revisiting EmbodiedQA: A Simple Baseline and Beyond. *IEEE Transactions on Image Processing* 29 (2020), 3984–3992. https://doi.org/10.1109/TIP.2020.2967584

[193] Kai Yuanqing Xiao, Logan Engstrom, Andrew Ilyas, and Aleksander Madry. 2021. Noise or Signal: The Role of Image Backgrounds in Object Recognition. https://openreview.net/forum?id=gl3D-xY7wLq

[194] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 176–188. https://doi.org/10.1145/3533767.3534220

[195] Xiaoyuan Xie, Joshua W.K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920 The Ninth International Conference on Quality Software.

[196] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 146–157. https://doi.org/10.1145/3293882.3330579

[197] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. DiffChaser: Detecting Disagreements for Deep Neural Networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 5772–5778. https://doi.org/10.24963/ijcai.2019/800

[198] Haoran Xu, Shuhui Fan, Yongjun Wang, Zhijian Huang, Hongzuo Xu, and Peidai Xie. 2020. Tree2tree Structural Language Modeling for Compiler Fuzzing. In *Algorithms and Architectures for Parallel Processing*, Meikang Qiu (Ed.). Springer International Publishing, Cham, 563–578.

[199] Haoran Xu, Yongjun Wang, Shuhui Fan, Peidai Xie, and Aizhi Liu. 2020. DSmith: Compiler Fuzzing through Generative Deep Learning Model with Attention. In *2020 International Joint Conference on Neural Networks (IJCNN)*. 1–9. https://doi.org/10.1109/IJCNN48605.2020.9206911

[200] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (apr 2023), 29 pages. https://doi.org/10.1145/3586049

[201] Ahmed Haj Yahmed, Houssem Ben Braiek, Foutse Khomh, Sonia Bouzidi, and Rania Zaatour. 2022. DiverGet: a Search-Based Software Testing approach for Deep Neural Network Quantization assessment. *Empir. Softw. Eng.* 27, 7 (2022), 193. https://doi.org/10.1007/s10664-022-10202-w

[202] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[203] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S. Huang. 2018. Generative Image Inpainting With Contextual Attention. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 5505–5514. https://doi.org/10.1109/CVPR.2018.00577

[204] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[205] Fuyuan Zhang, Sankalan Pal Chowdhury, and Maria Christakis. 2020. DeepSearch: a simple and effective blackbox attack for deep neural networks. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 800–812. https://doi.org/10.1145/3368089.3409750

[206] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P. Xing, Laurent El Ghaoui, and Michael I. Jordan. 2019. Theoretically Principled Trade-off between Robustness and Accuracy. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine*

Learning Research, Vol. 97), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7472–7482. http://proceedings.mlr.press/v97/zhang19p.html

[207] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2019.2962027

[208] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/3238147.3238187

[209] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Dai Ting. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA.

[210] Quan Zhang, Yifeng Ding, Yongqiang Tian, Jianmin Guo, Min Yuan, and Yu Jiang. 2021. AdvDoor: Adversarial Backdoor Attack of Deep Learning System. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 127–138. https://doi.org/10.1145/3460319.3464809

[211] Wei Emma Zhang, Quan Z. Sheng, Ahoud Alhazmi, and Chenliang Li. 2020. Adversarial Attacks on Deep-Learning Models in Natural Language Processing: A Survey. *ACM Trans. Intell. Syst. Technol.* 11, 3, Article 24 (apr 2020), 41 pages. https://doi.org/10.1145/3374217

[212] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards Characterizing Adversarial Defects of Deep Learning Software from the Lens of Uncertainty. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 739–751. https://doi.org/10.1145/3377811.3380368

[213] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System. In *43rd*

IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, 359–371. https://doi.org/10.1109/ICSE43902.2021.00043

[214] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3699–3715. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[215] Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. 2017. 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*. 2941–2951. https://www.aclweb.org/anthology/D17-1319

[216] Weimin Zhao, Sanaa Alwidian, and Qusay H. Mahmoud. 2022. Adversarial Training Methods for Deep Learning: A Systematic Review. *Algorithms* 15, 8 (2022). https://doi.org/10.3390/a15080283

[217] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-precision Weights. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HyQJ-mclg

[218] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning Deep Features for Discriminative Localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2921–2929. https://doi.org/10.1109/CVPR.2016.319

[219] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207.

[220] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic Testing of Driverless Cars. *Commun. ACM* 62, 3 (Feb. 2019), 61–67. https://doi.org/10.1145/3241979

[221] Michael Zhu and Suyog Gupta. 2018. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Sy1iIDkPM

[222] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: A Python Package For DNN Compression Research. (October 2019). https://arxiv.org/abs/1910.12232

[223] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8697–8710. https://doi.org/10.1109/CVPR.2018.00907